# Type-sensitive Control-flow Analysis

John Reppy
University of Chicago
jhr@cs.uchicago.edu

## Abstract

Higher-order typed languages, such as ML, provide strong support for data and type abstraction. While such abstraction is often viewed as costing performance, there are situations where it may provide opportunities for more aggressive program optimization. Specifically, we can exploit the fact that type abstraction guarantees representation independence, which allows the compiler to specialize data representations. This paper describes a first step in supporting such optimizations; namely a control-flow analysis that uses the program's type information to compute more precise results. We present our algorithm as an extension of Serrano's version of 0-CFA and we show that it respects types. We also discuss applications of the analysis with examples of optimizations enabled by the analysis that would not be possible using normal CFA.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Compilers, Optimization; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Abstract data types, Modules, packages

***General Terms*** Languages

***Keywords*** ML, Control-flow analysis

## 1. Introduction

One of distinguishing characteristics of the ML family of languages is the module system, which provides strong support for data and type abstraction, as well as for modular programming. While such abstraction is often been considered as a source of overhead that reduces program performance,[1] it has the important benefit of isolating clients of interfaces from implementation details, such as data representation. What is somewhat surprising is that, to the best of our knowledge, existing implementations of ML-like languages do not take advantage of abstractions in the source program to optimize data representations, *etc.* We are interested in such optimizations, particularly in the case of CML-style concurrency abstractions [Rep91, Rep99].

Control-flow information is not syntactically visible in higher-order languages, such as ML, so compilers must use *control-flow analysis* (CFA) algorithms to enable more advanced optimizations [Shi88]. One weakness of CFA is that one must be conservative about values that escape from the unit of analysis.[2] In this paper, we present a modification of Serrano's CFA [Ser95], that exploits type abstraction to improve the accuracy of analysis (and thus enable more optimizations). While our approach is presented in the context of a specific analysis algorithm, we believe that it is a general technique that can integrated into other flavors of flow analysis for higher-order languages. Unlike many program analyses, which are whole-program analyses, our analysis is modular by design.

Our analysis is based on the observation that if a type is abstract outside the scope of a module, then any value of that type must have been created inside the module at one of a statically known set of program points. Furthermore, the value can only be deconstructed at statically known program points in the module. We exploit this fact by computing for each abstract type an approximation of the values that escape the module into the wild, and thus might return. We use this approximation when confronted with an unknown value of the given type. To extend this mechanism to aggragate values and higher-order functions, we also use a more refined representation of *top* values that is indexed by type. For example, the result of calling an unknown function that returns a value of an abstract type can be approximated by the escaping values of that type. While these changes have the potential to significantly improve the quality of information provided by an analysis, they do not significantly add to the complexity of the implementation.

The remainder of this paper is organized as follows. In the next section, we briefly review Serrano's version of CFA, which serves as the basis of our algorithm, and give an example of how exploiting knowledge of type abstraction can produce a better result. Then, in Section 3, we present our type-sensitive analysis algorithm. We follow this with a small, but detailed, example of how the analysis works. In Section 5, we discuss the correctness of the analysis. We present our algorithm using a stripped down version of Core SML. In Section 6, we discuss how the algorithm can be extended to cover a greater fraction of ML features. Our motivation for this research is the optimization of CML-style concurrency mechanisms and we discuss that application as well as others in Section 7. We review related work in Section 8 and conclude in Section 9.

## 2. Background

In this section, we give a brief overview of Serrano's version of CFA [Ser95], which we use as the basis of our algorithm[3] and we motivate our improvements.

[1] For example, Andrew Appel reports that switching to a concrete representation of graphs in the Tiger compiler improved performance of liveness analysis by almost a factor of about ten [App98].

[2] One solution is whole-program analysis, but that does not scale well to large code bases.

[3] Our approach does not depend on the specifics of Serrano's algorithm and should be applicable to other versions of CFA.



*ML'06* September 16, 2006, Portland, Oregon, USA.

Serrano's analysis computes an imperative mapping $\mathcal{A}$ that maps variables to their approximate values. The approximate values are $\bot$, which denotes an approximation not yet computed, $\top$, which denotes an unknown value, and finite sets of function names. It is straightforward to extend this set of values with basic constants (*e.g.*, booleans and integers) and data constructors (*e.g.*, tuples).

A function is said to *escape* if it can be called at unknown sites. There are several ways that a function might escape: it may be exported by the module, it may be passed as an argument to an unknown function, or it may be returned as the result of an escaping function. If a function $f$ escapes, then $\mathcal{A}$ must be defined to map $f$'s parameters to $\top$, since we have no way of approximating them. But in a program that involves abstract types, we can exploit the type abstraction to compute better approximations. For example, consider the following SML code:

```
signature SIG =
  sig
    type t
    val new : int -> t
    val pick : t -> int
  end

structure M :> SIG =
  struct
    datatype t = C of int list
    fun new x = C[x]
    fun pick (C(y::_)) = y
  end
```

The programmer knows that the `pick` function can never raise a `Match` exception, even though it is defined using a non-exhaustive match. Without a whole-program analysis, existing compilers will not recognize this fact. Our analysis, however, uses the fact that type `t` is abstract, and will detect the fact that any value passed to `pick` as an argument must have been created by `new`.

## 3. Type-sensitive CFA

We are now ready to present our type-sensitive CFA algorithm. Our main modifications to Serrano's algorithm [Ser95] are:

- we use a more refined representation of approximate values
- we compute an additional approximation $\mathcal{T}$ that tracks escaping values of abstract type.
- our algorithm is presented as a functional program without side effects.

### 3.1 Preliminaries

We present our algorithm in the context of a small typed language. This language is a monomorphic subset of Core SML [MTHM97] with explicit types[4]. Standard ML and other ML-like languages use modules to organize code and signature ascription to define abstraction. For this paper, we use the **abstype** definition to define abstractions in lieu of modules. We further simplify this definition form to only have a single data constructor. Figure 1 gives the abstract syntax for this simple language. The top-level **abstype** definition is the unit of analysis (a program consists of a collection of these). Each **abstype** definition defines a new abstract type ($T$) and corresponding data constructor ($C$) and a collection of functions ($fb_i$). Outside the **abstype** definition, the type $T$ is abstract (*i.e.*, the data constructor $C$ is not in scope). The expression forms include let-bindings, nested function bindings, function application, data-constructor application and deconstruction,[5] and pair construction and projection. Types include abstract types (T), function types, and pair types. Abstract types are either predefined types (*e.g.*, `unit`, `int`, `bool`, *etc.*) or are defined by an **abstype** definition.

$$
\begin{array}{rcl}
p & ::= & e \\
 & | & d;\ p \\
\\
d & ::= & \textbf{abstype}\ T = C\ \textbf{of}\ \tau\ \textbf{with}\ fb_1\ \cdots\ fb_n\ \textbf{end} \\
\\
fb & ::= & \textbf{fun}\ f\ (x) = e \\
\\
e & ::= & x \\
 & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
 & | & \textbf{fun}\ f\ (x) = e_1\ \textbf{in}\ e_2 \\
 & | & e_1\ e_2 \\
 & | & C\ e \\
 & | & \textbf{let}\ C\ x = e_1\ \textbf{in}\ e_2 \\
 & | & \langle e_1, e_2 \rangle \\
 & | & \textbf{fst}(e) \\
 & | & \textbf{snd}(e) \\
\\
\tau & ::= & \mathrm{T} \\
 & | & \tau_1 \rightarrow \tau_2 \\
 & | & \tau_1 \times \tau_2
\end{array}
$$

**Figure 1.** A simple language

We assume that variables, abstract-type names, and data-constructor names are globally unique. We also assume that variables and constructors are annotated with their type. We omit this type information most of the time for the sake of brevity, but, when necessary, we write it as a superscript (*e.g.*, $x^\tau$). One should think of this language as a compiler's intermediate representation produced by the typechecking phase.

### 3.2 Approximate values

The analysis computes a mapping from variables to approximate values, which are given by the following grammar:

$$
\begin{array}{rcl}
\widehat{v} & ::= & \bot \\
 & | & C(\widehat{v}) \\
 & | & \langle \widehat{v}_1, \widehat{v}_2 \rangle \\
 & | & F \\
 & | & \widehat{T} \\
 & | & \widehat{\tau_1 \rightarrow \tau_2} \\
 & | & \top
\end{array}
$$

where $C$ is a data constructor, $F$ is a finite set of function names, and $T$ is an abstract type. We use $\bot$ to denote undefined or not yet computed values, $C(\widehat{v})$ for an approximate value constructed by applying $C$ to $\widehat{v}$, $\langle \widehat{v}_1, \widehat{v}_2 \rangle$ for an approximate pair, and $F$ for a set of known functions. Our analysis will only compute sets of functions $F$ where all the members have the same type (see Section 5 for a proof of this property) and so we extend our type annotation syntax to include such sets. In addition to the single *top* value found in most presentations of CFA, we have a family of top values ($\widehat{\tau}$) indexed by type. The value $\widehat{\tau}$ represents an unknown value of type $\tau$ (where $\tau$ is either a function or abstract type). The auxiliary function $\mathcal{U} : \textsc{Type} \rightarrow \textsc{Value}$ maps types to their corresponding

[4] We discuss handling polymorphic types and user-defined type constructors in Section 6.

[5] In a language with sum types, deconstruction would be replaced by a case expression.

unknown value:

$$\begin{array}{rcl} \mathcal{U}(\mathrm{T}) & = & \widehat{T} \\ \mathcal{U}(\tau_1 \to \tau_2) & = & \widehat{\tau_1 \to \tau_2} \\ \mathcal{U}(\langle \tau_1, \tau_2 \rangle) & = & \langle \mathcal{U}(\tau_1), \mathcal{U}(\tau_n) \rangle \end{array}$$

Note that for unknown pair values, we preserve the fact that they have a pair structure. Lastly, the $\top$ value is used to cutoff expansion of recursive types as described below.

We define the *join* of two approximate values as follows:

$$\begin{array}{rcl} \bot \vee \widehat{v} & = & \widehat{v} \\ \widehat{v} \vee \bot & = & \widehat{v} \\ C(\widehat{v}_1) \vee C(\widehat{v}_2) & = & C(\widehat{v}_1 \vee \widehat{v}_2) \\ \langle \widehat{v}_1, \widehat{v}_2 \rangle \vee \langle \widehat{v'}_1, \widehat{v'}_2 \rangle & = & \langle \widehat{v}_1 \vee \widehat{v'}_1, \widehat{v}_2 \vee \widehat{v'}_2 \rangle \\ F \vee F' & = & F \cup F' \\ \top \vee \widehat{v} & = & \top \\ \widehat{\tau} \vee \widehat{v} & = & \widehat{\tau} \\ \widehat{v} \vee \widehat{\tau} & = & \widehat{\tau} \\ \widehat{v} \vee \top & = & \top \end{array}$$

Note that this operation is not total, but it is defined for any two approximate values of the same type and we show in Section 5 that it preserves types.

One technical complication is that we need to keep our approximate values finite. For example, consider the following pathological example:

$$\textbf{abstype}\ T = C\ \textbf{of}\ T\ \textbf{with fun}\ f\,(x) = C\,x\ \textbf{end}$$

If we are not careful, our analysis might diverge computing ever larger approximations of $C^\infty(\bot)$ as the result of $f$. To avoid this problem, we define a limit on the depth of approximations for recursive types as follows:

$$\begin{array}{rcl} \lceil \bot \rceil_{\mathbf{C}} & = & \bot \\ \lceil C^{\tau \to T}(\widehat{v}) \rceil_{\mathbf{C}} & = & \begin{cases} \top & \text{if } C \in \mathbf{C} \\ C(\lceil \widehat{v} \rceil_{\mathbf{C} \cup \{C\}}) & \text{if } C \notin \mathbf{C} \end{cases} \\ \lceil \langle \widehat{v}_1, \widehat{v}_2 \rangle \rceil_{\mathbf{C}} & = & \langle \lceil \widehat{v}_1 \rceil_{\mathbf{C}}, \lceil \widehat{v}_2 \rceil_{\mathbf{C}} \rangle \\ \lceil F \rceil_{\mathbf{C}} & = & F \\ \lceil \widehat{\tau} \rceil_{\mathbf{C}} & = & \widehat{\tau} \end{array}$$

where $\mathbf{C} \subset$ DATACON is a set of constructors. We write $\lceil \widehat{v} \rceil$ for $\lceil \widehat{v} \rceil_\emptyset$. We use $\top$ to cutoff the expansion of approximate values instead of $\widehat{T}$, because the approximation of escaping values of type $T$ may not be an accurate approximation of the nested values. This definition does not allow nested applications of the same constructor. For example, the analysis will be forced to approximate the escaping values of type $T$ by $C(\top)$ in the above example.

### 3.3 CFA

Our analysis algorithm computes a triple of approximations: $\mathcal{A} = (\mathcal{V}, \mathcal{R}, \mathcal{T})$, where

| | | | |
|---|---|---|---|
| $\mathcal{V}$ | $\in$ | $\text{VAR} \to \widehat{\text{VALUE}}$ | variable approximation |
| $\mathcal{R}$ | $\in$ | $\text{FUNID} \to \widehat{\text{VALUE}}$ | function-result approximation |
| $\mathcal{T}$ | $\in$ | $\text{ABSTY} \to \widehat{\text{VALUE}}$ | escaping abstract-value approximation |

Our $\mathcal{V}$ approximation corresponds to Serrano's $\mathcal{A}$. The $\mathcal{R}$ approximation records an approximation of function results for each known function; this approximation is used in lieu of analysing a function's body when the function is already being analysed and is needed to guarantee termination. We use the $\mathcal{T}$ approximation to interpret abstract values of the form $\widehat{T}$.

We present the analysis algorithm using SML syntax extended with mathematical notation such as set operations, and the $\vee$ operation on approximate values. We use the notation $[\![e]\!]$ to denote that "$e$" is an object-language syntactic form and $\mathcal{V}[x \mapsto v]$ to denote the functional update of an approximation (likewise for $\mathcal{R}$ and $\mathcal{T}$).

Our unit of analysis is a single **abstype** definition. We use LVAR to denote the set of variables defined in the definition, GVAR to denote variables defined elsewhere, and VAR $=$ LVAR $\cup$ GVAR for all variables defined or mentioned in the definition being analyzed. We denote the known function identifiers by FUNID $\subset$ LVAR (*i.e.*, those variables that are defined by function bindings). These include the top-level function bindings in the definition, as well as any nested function definitions. Our algorithm analyses the function definitions in the declaration repeatedly until a fixed-point is reached. The initial approximations map local variables, function results, and abstract types to $\bot$, and map global variables and external types to unknown values.

```
fun cfa [[abstype T = C of τ with fb_1 ··· fb_n end]] =
      let
      fun iterate A_0 = let
            val A_1 = cfaFB (A_0, fb_1)
            ···
            val A_n = cfaFB (A_{n-1}, fb_n)
            in
              if (A_0 ≠ A_n)
                then iterate A_n
                else A_0
            end
      let V = {x ↦ ⊥ | x ∈ LVAR \ FUNID}
              ∪ {f ↦ {f} | f ∈ FUNID}
              ∪ {x ↦ U(τ) | x^τ ∈ GVAR}
      let R = {f ↦ ⊥ | f ∈ FUNID}
      let T = {T ↦ ⊥} ∪ {S ↦ Ŝ | S ∈ (ABSTY \ {T})}
      in
        iterate (V, R, T)
      end
```

The `cfaFB` function analyses a function binding in the **abstype** definition by "applying" the function to the unknown value of the function's argument type. The result of the application is then recorded as escaping.

```
fun cfaFB (A, [[fun f(x^τ) = e]]) = let
      val (A, v̂) = applyFun ({}, A, f, U(τ))
      in
        escape ({}, A, v̂)
      end
```

The `applyFun` function analyses the application of a known function $f$ to an approximate value $\widehat{v}$. The first argument to `applyFun` is a set $\mathbf{M} \in 2^{\text{FUNID}}$ of known functions that are currently being analysed; if $f$ is in this set, then we use the approximation $\mathcal{R}$ instead of recursively analysing the $f$'s body. This mechanism is necessary to guarantee termination when analysing recursive functions. We assume the existence of a helper function `bindingOf` that maps known function names to their bindings in the source. Once we have computed the approximate result ($r$) of evaluating the function's body, we add that information to the result approximation.

```
fun applyFun (M, A as (V,R,T), f, v̂) =
      if f ∈ M
        then (A, R(f))
        else let
          val [[fun f(x) = e]] = bindingOf (f)
          val V = V[x ↦ ⌈V(x) ∨ v̂⌉]
          val ((V, R, T), r) =
                cfaExp (M ∪ {f}, (V,R,T), [[e]])
          val R = R[f ↦ ⌈R(f) ∨ r⌉]
          in
            ((V,R,T), r)
          end
```

```
fun cfaExp (M, A as (V,R,T), [[x]]) = (A, V(x))
  | cfaExp (M, A, [[let x = e1 in e2]]) = let
      val ((V,R,T), v̂) = cfaExp (M, A, [[e1]])
      val V = V[x ↦ ⌈V(x) ∨ v̂⌉]
      in
        cfaExp (M, (V, R, T), [[e2]])
      end
  | cfaExp (M, A, [[fun f(x) = e1 in e2]]) =
      cfaExp (M, A, [[e2]])
  | cfaExp (M, A, [[e1 e2]]) = let
      val (A, v̂1) = cfaExp (M, A, [[e1]])
      val (A, v̂2) = cfaExp (M, A, [[e2]])
      in
        apply (M, A, v̂1, v̂2)
      end
  | cfaExp (M, A, [[C e]]) = let
      val (A, v̂) = cfaExp (M, A, [[e]])
      in
        (A, C(v̂))
      end
  | cfaExp (M, A, [[let C x = e1 in e2]]) = let
      val ((V,R,T), v̂) = cfaExp (M, A, [[e1]])
      val V = decon (V, T, [[C x]], v̂)
      in
        cfaExp (M, (V,R,T), [[e2]])
      end
  | cfaExp (M, A, [[⟨e1, e2⟩]]) = let
      val (A, v̂1) = cfaExp (M, A, [[e1]])
      val (A, v̂2) = cfaExp (M, A, [[e2]])
      in
        (A, ⟨v̂1, v̂2⟩)
      end
  | cfaExp (M, A, [[fst(e)]]) = (
      case cfaExp (M, A, [[e]])
       of (A, ⟨v̂1, v̂2⟩) => (A, v̂2)
        | (A, v̂) => (A, v̂)
      (* end case *))
  | cfaExp (M, A, [[snd(e)]]) = (
      case cfaExp (M, A, [[e]])
       of (A, ⟨v̂1, v̂2⟩) => (A, v̂1)
        | (A, v̂) => (A, v̂)
      (* end case *))
```

**Figure 2.** CFA for expressions

The `escape` function records the fact that a value escapes into the wild. If the value has an abstract type, then it is added to the approximation of wild values for the type; if it is a set of known functions, then we apply each function in the set to the appropriate top value; and if it is a pair, we record that its subcomponents are escaping. The `escape` function also takes the set of currently active functions as its first argument, since it needs to pass this value to the `applyFun` function.

```
fun escape (_, (V,R,T), C(v̂)) =
      (V, R, T[T ↦ ⌈T(T) ∨ C(v̂)⌉])
  | escape (M, A, F) = let
      fun esc (f^τ1 → τ2, A) = let
            val (A, v̂) = applyFun(M, A, f, U(τ1))
            in A end
      in
        fold esc A F
      end
  | escape (M, A, ⟨v̂1, , v̂2⟩) = let
      val A1 = escape (M, A, v̂1)
      val A2 = escape (M, A1, v̂2)
      in A2 end
  | escape (_, A, v̂) = A
```

Expressions are analysed by the `cfaExp` function, whose code is given in Figure 2. This function takes the set of active functions, an approximation triple, and an syntactic expression as arguments and returns updated approximations and a value that approximates the result of the expression. For function applications, we use the `apply` helper function (discussed below) and for value deconstruction, we use the `decon` helper function, which handles the deconstruction of approximate values and their binding to variables. When the value is unknown (*i.e.*, $\widehat{T}$), then we use the $\mathcal{T}$ approximation to determine the value being deconstructed.

```
fun decon (V, T, [[C(x)]], C(v̂)) = V[x ↦ ⌈V(x) ∨ v̂⌉]]
  | decon (V, T, [[C^τ→T(x)]], T̂) = (case T(T)
       of T̂ => V[x ↦ ⌈V(x) ∨ U(τ)⌉]
        | v̂ => decon(V, T, [[C(x)]], v̂)
      (* end case *))
  | decon (V, T, [[C(x)]], ⊥) = V
  | decon (V, T, [[C(x)]], ⊤) = V[x ↦ ⊤]
```

The `apply` function records the fact that an approximate function value is being applied to a approximate argument. When the approximation is a set of known functions, then we apply each function in the set to the argument compute the join of the results. When the function is unknown (*i.e.*, a top value), then the argument is marked as escaping and the result is the top value for the function's range.

```
fun apply (M, A, F, arg) = let
      fun applyf (f, (A, res)) = let
            val (A, v̂) = applyFun (M, A, f, arg)
            in
              (A, res ∨ v̂)
            end
      in
        fold applyf (V, T) F
      end
  | apply (M, A, τ1 →^ τ2, v̂) = let
      val A = escape(M, A, v̂)
      in
        (A, τ̂2)
      end
  | apply (M, A, ⊥, v̂) = (A, ⊥)
  | apply (M, A, ⊤, v̂) = let
      val A = escape(M, A, v̂)
      in
        (A, ⊤)
      end
```

## 4. An example

To illustrate the analysis, we step through its application to the following small example:[6]

```
abstype t = C of (int * int)
  with
    fun new x = C(1, x)
    fun pick y = let C(z) = y in fst(z)
  end
```

This code has two functions and three other local variables:

$$\begin{aligned} \textsc{FunId} &= \{\texttt{new}, \texttt{pick}\} \\ \textsc{LVar} &= \{\texttt{x}, \texttt{y}, \texttt{z}\} \cup \textsc{FunId} \\ \textsc{GVar} &= \{\} \end{aligned}$$

[6] This example is a variation of the one given in Section 2. We have also taken the liberty of adding integer constants to our language and to the representation of approximate values.

The initial approximations are

$$\begin{aligned}\mathcal{A}_0 &= (\mathcal{V}_0, \mathcal{R}_0, \mathcal{T}_0)\\ \mathcal{V}_0 &= \{\texttt{new} \mapsto \{\texttt{new}\}, \texttt{pick} \mapsto \{\texttt{pick}\}\}\\ &\quad \cup \{\texttt{x} \mapsto \bot, \texttt{y} \mapsto \bot, \texttt{z} \mapsto \bot\}\\ \mathcal{R}_0 &= \{f \mapsto \bot \mid f \in \textsc{FunID}\}\\ \mathcal{T}_0 &= \{t \mapsto \bot\} \cup \{S \mapsto \widehat{S} \mid S \neq t\}\end{aligned}$$

The `cfa` function will apply `cfaFB` to each of the bindings. We start with `new` and compute

```
(A1, v̂1) = applyFun({}, A0, new, int^)
```

Computing this application of `applyFun` involves computing

```
cfaExp ({new}, (V1, R0, T0), [[C(1, x)]])
```

where $\mathcal{V}_1 = \mathcal{V}_0[\texttt{x} \mapsto \widehat{\texttt{int}}]$. The result of analyzing the body of `new` will be $((\mathcal{V}_1, \mathcal{R}_0, \mathcal{T}_0), C(1, \widehat{\texttt{int}}))$, thus we have

$$\begin{aligned}\mathcal{A}_1 &= (\mathcal{V}_1, \mathcal{R}_1, \mathcal{T}_0)\\ \mathcal{R}_1 &= \{\texttt{new} \mapsto C(1, \widehat{\texttt{int}}), \texttt{pick} \mapsto \bot\}\\ \widehat{v}_1 &= C(1, \widehat{\texttt{int}})\end{aligned}$$

The last step in analyzing `new` is to compute

```
escape ({}, A1, v̂1)
```

which results in an enriched approximation that records the escaping value of type `t`.

$$\begin{aligned}\mathcal{A}_2 &= (\mathcal{V}_1, \mathcal{R}_1, \mathcal{T}_1)\\ \mathcal{T}_1 &= \{\texttt{t} \mapsto \texttt{C}(1, \widehat{\texttt{int}})\}\end{aligned}$$

Now we are ready to analyse the `pick` function binding, which means computing

```
(A3, v̂3) = applyFun({}, A2, pick, t̂)
```

Let $\mathcal{V}_2 = \mathcal{V}_1[\texttt{y} \mapsto \widehat{\texttt{t}}]$, then we evaluate the body of `pick` with the initial approximations $(\mathcal{V}_2, \mathcal{R}_1, \mathcal{T}_1)$. The interesting case is when `cfaExp` gets to the deconstruction of the value bound to `y`. In this case, we must compute

```
decon (V2, T1, [[C(z)]], t̂)
```

This case is handled by the second clause of the `decon` function, which applies $\mathcal{T}_1$ to `t`, producing $\texttt{C}(1, \widehat{\texttt{int}})$, which results in the recursive call of `decon`

```
decon (V2, T1, [[C(z)]], C(1, int^))
```

The recursive call is handled by the first clause of `decon`, which returns the augmented value approximation

$$\mathcal{V}_3 = \mathcal{V}_2[\texttt{z} \mapsto (1, \widehat{\texttt{int}})]$$

This approximation will be used in the analysis of $[\![\mathbf{fst}(\texttt{z})]\!]$, which will produce 1 as its approximate result. Thus, the result of `applyFun` on `pick` is

$$\begin{aligned}\mathcal{A}_3 &= (\mathcal{V}_2, \mathcal{T}_1, \mathcal{R}_2)\\ \mathcal{R}_2 &= \mathcal{R}_1[\texttt{pick} \mapsto 1]\\ \widehat{v}_3 &= 1\end{aligned}$$

The `escape` function will not change the approximations in this case, so $\mathcal{A}_3$ is the result of the first iteration over the function bindings. It is also the fixedpoint of the analysis for this example, so the final result is:

$$\begin{aligned}\mathcal{V}(\texttt{x}) &= \widehat{\texttt{int}}\\ \mathcal{V}(\texttt{y}) &= \texttt{C}(1, \widehat{\texttt{int}})\\ \mathcal{V}(\texttt{z}) &= (1, \widehat{\texttt{int}})\\ \mathcal{R}(\texttt{new}) &= \texttt{C}(1, \widehat{\texttt{int}})\\ \mathcal{R}(\texttt{pick}) &= 1\\ \mathcal{T}(\texttt{t}) &= \texttt{C}(1, \widehat{\texttt{int}})\end{aligned}$$

## 5. Correctness of the analysis

The correctness of our analysis can be judged on several dimensions. First, there is the question of safety: does the analysis compute an approximation of the actual computation? Second is the question of whether the algorithm terminates? The third correctness issue is the question of whether the approximations computed by the analysis are faithful to the type of the program. This question is important, since our analysis is guided by type information in a number of situations. We discuss the first and third of these questions in the remainder of this section.

### 5.1 Safety

We postulate that our analysis is *safe*, *i.e.*, that if it computes a value approximation $\mathcal{V}$, then for any variable $x \in \mathrm{dom}(\mathcal{V})$ and any execution of the program, the values taken on by $x$ will be covered by the approximate value $\mathcal{V}(x)$. One can formalize this statement in terms of a collecting semantics [CC77, Shi91] and prove it using standard techniques, but we will make a less formal argument here. The core of our analysis is the well known 0-CFA, which has been proven correct in a number of papers, but we have extended this analysis with the $\mathcal{T}$ approximation for tracking abstract values that escape to the wild. For a given abstract-type definition

```
abstype T = C of τ in ... end
```

the ML type system restricts the scope of `C` to the "`...`"; thus, values of type `T` can only be constructed/deconstructed in the body of the definition. Therefore, we claim that if the $\mathcal{T}$ approximation computed by our analysis is "*safe*," then our analysis is correct.

There are two aspects to the safety of $\mathcal{T}$: first, does it correctly approximate the values that escape and second, does the analysis correctly identify all possible places where escaping values could reenter the definition? There are only two ways for values to escape the definition: they can be returned in the result of one of the operations or they can be passed as an argument to an external or unknown function.[7] The first of these cases is covered by the call to `escape` in `cfaFB`, while the second is covered by the call to `escape` in in `apply`. Thus, we claim that any escaping abstract value in any possible execution will be covered by the $\mathcal{T}$ approximation computed by our analysis. There are also only two ways for escaping values to enter the definition: they can be passed in an argument to one of the definition's operations or they can be returned by a call to an external or unknown function. In both of these situations, we use the approximate value $\mathcal{U}(\tau)$ to represent unknown values of type $\tau$. If such a value propagates to a deconstruction site, then we use the $\mathcal{T}$ approximation to determine the values bound to the pattern variables.

### 5.2 Termination

The question of termination has been addressed by Serrano [Ser95] (the bounding of the sizes of abstract values is crucial to the termi-

[7] If our language had references, then assignment would be another way for values to escape.

$$\frac{\vdash fb_1 : \mathbf{Ok} \quad \cdots \quad \vdash fb_n : \mathbf{Ok}}{\vdash \mathbf{abstype}\ T = C^{\tau \to T}\ \mathbf{of}\ \tau\ \mathbf{with}\ fb_1\ \cdots\ fb_n\ \mathbf{end} : \mathbf{Ok}}$$

$$\frac{\vdash e : \tau_2}{\vdash \mathbf{fun}\ f^{\tau_1 \to \tau_2}\,(x^{\tau_1}) = e : \mathbf{Ok}}$$

$$\frac{}{\vdash x^{\tau} : \tau}$$

$$\frac{\vdash e_1 : \tau_1 \qquad \vdash e_2 : \tau_2}{\vdash \mathbf{let}\ x^{\tau_1} = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\frac{\vdash \mathbf{fun}\ f\,(x) = e_1 : \mathbf{Ok} \qquad \vdash e_2 : \tau}{\vdash \mathbf{fun}\ f\,(x) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\frac{\vdash e_1 : \tau_2 \to \tau \qquad \vdash e_2 : \tau_2}{\vdash e_1\ e_2 : \tau}$$

$$\frac{\vdash e : \tau}{\vdash C^{\tau \to T}\ e : T}$$

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : \tau_2}{\vdash \mathbf{let}\ C^{\tau \to T}\ x^{\tau} = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\frac{\vdash e_1 : \tau_1 \quad \vdash e_2 : \tau_2}{\vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\vdash e : \tau_1 \times \tau_2}{\vdash \mathbf{fst}(e) : \tau_1}$$

$$\frac{\vdash e : \tau_1 \times \tau_2}{\vdash \mathbf{snd}(e) : \tau_2}$$

**Figure 3.** Typing rules for declarations, function bindings, and expressions

nation of the fixed-point iteration) and his argument applies directly to our version of his algorithm.

A related question is does the analysis avoid uncaught runtime exceptions? The two helper functions `decon` and `apply` are both partial, so the possibility of a `Match` exception exists (for the fourth argument of `decon` and the third argument of `apply`). We claim that such exceptions will not occur. Because the program being analyzed is type correct and based on the type correctness of the analysis (discussed in the next section), we see that any type-correct approximate value that might be passed in as an argument is covered by the functions' cases.

### 5.3 Type correctness of the analysis

Since we assume that programs have already been typechecked and that variables and data-constructors are annotated with their types, the typing rules for our language do not require a context. We have three judgment forms:

| | |
|---|---|
| $\vdash d : \mathbf{Ok}$ | the declaration $d$ is well-typed |
| $\vdash fb : \mathbf{Ok}$ | the function binding *fb* is well-typed |
| $\vdash e : \tau$ | the expression $e$ is well-typed and has type $\tau$ |

The typing rules for our language are straightforward and are given in Figure 3.

We also define typing rules for approximate values (again annotations take the place of context). These rules are given in Figure 4. Note that the $\bot$ and $\top$ values can have any well-formed type.

We say that a variable approximation $\mathcal{V}$ is *type consistent* with respect to a code fragment (*i.e.*, declaration, function binding, or expression), if for all variables $x^{\tau}$ defined or mentioned in the

$$\frac{\vdash \tau : \mathbf{Type}}{\vdash \bot : \tau}$$

$$\frac{\vdash \widehat{v} : \tau}{\vdash C^{\tau \to T}(\widehat{v}) : T}$$

$$\frac{\vdash \widehat{v}_1 : \tau_1 \quad \vdash \widehat{v}_2 : \tau_2}{\vdash \langle \widehat{v}_1, \widehat{v}_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\vdash f : \tau \to \tau' \quad \textit{for all } f \in F}{\vdash F : \tau \to \tau'}$$

$$\frac{}{\vdash \widehat{\tau} : \tau}$$

$$\frac{\vdash \tau : \mathbf{Type}}{\vdash \top : \tau}$$

**Figure 4.** Typing rules for approximate values

fragment, $x \in \mathrm{dom}(\mathcal{V})$ and $\vdash \mathcal{V}(x) : \tau$. A function-result approximation $\mathcal{R}$ is *type consistent* with respect to a fragment if for all functions $f^{\tau_1 \to \tau_2}$ defined in the fragment, $f \in \mathrm{dom}(\mathcal{R})$ and $\vdash \mathcal{R}(f) : \tau_2$. Likewise, an approximation $\mathcal{T}$ is *type consistent* with respect to a fragment if for all type names defined or mentioned in the fragment, $T \in \mathrm{dom}(\mathcal{T})$ and $\vdash \mathcal{T}(T) : T$. An approximation triple $\mathcal{A} = (\mathcal{V}, \mathcal{R}, \mathcal{T})$ is type consistent if its components are type consistent.

Our main result is that `cfaExp` computes type correct approximations (or *respects types*), but we need the following lemma first, which says that the $\vee$ operator preserves types.

LEMMA 1. *If* $\vdash \widehat{v}_1 : \tau$ *and* $\vdash \widehat{v}_2 : \tau$, *then* $\vdash \widehat{v}_1 \vee \widehat{v}_2 : \tau$.

***Proof.*** *The proof two parts. First, we can prove that if* $\vdash \widehat{v}_1 : \tau$ *and* $\vdash \widehat{v}_2 : \tau$, *then* $\widehat{v}_1 \vee \widehat{v}_2$ *is defined (recall that* $\vee$ *is partial) by case analysis of the shape of* $\tau$ *and the typing rules. Then, a simple inductive argument shows that* $\vdash \widehat{v}_1 \vee \widehat{v}_2 : \tau$

THEOREM 1. *Given an expression e, with* $\vdash e : \tau$, *and a type consistent approximation triple* $\mathcal{A}$, *if* $\texttt{cfaExp}(\{\}, \mathcal{A}, e)$ *returns the result* $(\mathcal{A}', \widehat{v})$, *then* $\mathcal{A}'$ *is type consistent and* $\vdash \widehat{v} : \tau$.

***Proof.*** *This theorem is proven by induction on the structure of the expressions. The induction is well-founded because the analysis never examines a function's body if it is already being analysed.*

Since the initial approximations in `cfa` are type consistent, the above theorem guarantees that the resulting approximation will be type consistent.

## 6. Extensions

We presented our algorithm as a modification of Serrano's 0-CFA, but we see no reason why our techniques will not extend to similar program analyses, such as sub-zero CFA [AD98] or 1-CFA [Shi91].

We have also used a greatly reduced subset of Core SML to present our approach. In the remainder of this section, we discuss extensions of the analysis to other ML features.

### 6.1 Modules

Our example language uses the **abstype** declaration as its unit of modularity and analysis. This feature has been depreciated in SML in favor of using modules and signature ascription, so to extend our technique to the full ML language, we need to have a different way of identifying abstract types. The basic goal is to have the $\mathcal{T}$ approximation return $\mathcal{U}(T)$ when $T$ is not abstract in

the module's signature, since clients of the module can construct values of type $T$ using the constructor $C$. We see two ways to achieve this behavior. The first is to make the initialization of the $\mathcal{T}$ approximation map depend on the module's signature. If a datatype has the specification

```
datatype T = C of τ
```

in the signature, then we initialize $\mathcal{T}(T)$ to $\mathcal{U}(T)$, instead of the $\bot$.

The other approach would be to view the constructors of $T$ as escaping functions, which has the same effect of $\mathcal{T}$. This approach also handles the situation where a constructor is passed as a value to an external function.

A more difficult issue is that abstract types can be defined without data constructors using opaque signature matching. For example,

```
structure S :> sig
    type set
    val singleton : int -> set
    val items : set -> int list
    val member : (set * int) -> bool
    ...
  end = struct
    type set = int list
    fun singleton x = [x]
    fun items s = s
    fun member (s, x) =
          List.exists (fn y => (x = y)) s
    ...
  end
```

In this situation there are no syntactic signposts to mark the type abstractions and some uses of the type "`int list`" are abstract and some are concrete. One possible solution is to use the module's signature as a guide to add annotations to the module's code that mark the abstractions. This process is similar to Leroy's *type-directed unboxing* [Ler92] and Grossman, *et al.*'s *syntatic type abstraction* [GMZ00]. For example, the above code would become

```
fun singleton x = ABS[x]
fun member (ABS s, x) =
      List.exists (fn y => (x = y)) s
```

where `ABS` marks the abstraction barriers. As is the case with Leroy's technique, aggregates of abstract values are problematic. For example, if we had

```
val union : set list -> set
```

in the signature with the implementation

```
fun union ss = List.concat ss
```

then we would need to map the projection of `ABS` over the argument list

```
fun union ss = let
      val ss' = List.map (fn ABS s => s) ss
      in
        ABS(List.concat ss')
      end
```

which introduces unnecessary computation and reduces the quality of the analysis. More thought on this problem is definitely required.

### 6.2 Polymorphism and type constructors

We presented our analysis for a language without polymorphism or user-defined type constructors. The main impact of extending it to a richer type system is the representation of top values. The $\mathcal{U}$ function gets extended as follows:

$$\begin{aligned} \mathcal{U}(\alpha) &= \top \\ \mathcal{U}(\overrightarrow{\tau}\,T) &= \widehat{\overrightarrow{\tau}\,T} \end{aligned}$$

We also extend the domain of the $\mathcal{T}$ approximation to include abstract-type constructor applications. We believe that tracking each distinct application of an abstract-type constructor will be useful in optimizing modules that use phantom types in their interfaces.

Polymorphism is also a form of type abstraction, which we should exploit when analyzing calls to external or unknown functions. For example, the `map` function on lists has the type

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

From this type, we know that the `map` function can only manipulate the items in the argument and result lists by using its first argument. Thus, a call to map like the one for the `union` function in the previous section

```
List.map (fn ABS s => s) ss
```

does not actually expose the representation of the abstract values.

### 6.3 Datatypes

The **abstype** declarations in our language are limited to single constructors. Extending the language with ML datatypes suggests a corresponding extension of approximate values to include sets of tagged values. The analysis then takes on some of the aspects of shape analysis [NNH99, Section 2.6] and one might want a cutoff on recursive values that is more generous.

## 7. Applications

Serrano's paper [Ser95] describes a number of applications of CFA in a SCHEME compiler. Many of these applications, such as identifying known functions, are also useful in ML compilers. Our algorithm will provide as good, or better information without a significant increase in implementation complexity. Furthermore, there are situations where we believe our techniques are necessary to achieve the desired optimizations. These situations can be divided into roughly two kinds. The first are situations where we want to identify all uses of a value, which can allow us to specialize the value's representation. For example, data structures might be flattened, which reduces space and improves access time, and fields might be packed (*e.g.*, storing booleans in a byte instead of a word), which reduces space. One can also reorganize the order of fields to improve cache performance [CDL99]. The second case is when we want to identify the possible definitions of an argument to an operation, which can allow us to specialize the operation. For example, we may be able to eliminate conditionals or use more efficient calling conventions. In both situations, if the value is embedded in an abstraction, we can track its uses, even if it escapes its defining module. We give an example of each of these situations below.

### 7.1 Optimizing CML

Our original motivation for this algorithm was the problem of optimizing Concurrent ML (CML) [Rep99]. Our goal is to statically determine the usage patterns of communication primitives (we call these patterns the *communication topology*), which will allow us to specialize them to more efficient versions. A *known channel* is a channel for which we know all of its use sites (*i.e.*, all of the `send` and `recv` operations). Our analysis can compute a static approximation of the communication topology of known channels, but to be effective, we need the type-sensitive CFA described in this paper to determine the known channels.[8]

For example, consider the simple service implemented in Figure 5. The `new` function creates a new service, which is represented

[8] Another analysis technique might also work, but our type-sensitive CFA fits well with common CML programming idioms.

```
abstype serv = S of (int * int chan) chan
  in
    fun new () = let
          val reqCh = channel()
          fun server v = let
                val (req, replCh) = recv reqCh
                in
                  send(replCh, v);
                  server req
                end
          in
            spawn (server 0);
            S reqCh
          end
    fun call (S ch, v) = let
          val replCh = channel()
          in
            send (ch, (v, replCh));
            recv replCh
          end
  end
```

**Figure 5.** A simple service with an abstract client-server protocol

by an abstract type, and the `call` function allows clients to request the service. We have developed an analysis that can detect that the server's request channel (`reqCh`) is used by potentially many different senders, but by only on receiver, and that the reply channel allocated for a given request (`replCh`) is used only once [Xia05, RX06]. These properties allow the optimizer to replace the general channel operations with more specialized ones.

Continuing with the example, Figure 6 illustrates the flow of the server's request channel from its allocation site in the `new` function to its two use sites (the `recv` in `new` and the `send` in `call`). Note that while the unknown clients of the service may store a a `serv` value in data structures, *etc.*., they may not access the internal representation and perform operations directly on the request channel. Because the server's request channel is embedded in an abstract value, our analysis is able to determine its use sites and we can classify it as a known channel. The analysis will also identify `replCh` as a known channel, which allows the communication topology of this example to be accurately approximated.

### 7.2 Perlin noise

Procedural generation of geometry and textures in computer graphics applications often uses *noise* functions to introduce variability and produce more realistic images. One popular technique, which was developed by Ken Perlin, defines a mapping from $\Re^n$ to the unit interval, such that values that are close in $\Re^n$ will be mapped to values that are close (i.e., the noise has local similarity, but global randomness) [PH89, Per02]. To compute the noise function, we first divide up a unit $n$-cube into equal sized cells and pre-compute a random gradient vector in $\Re^n$ for each of the cell corners. Then the noise function is computed by mapping a point in $\Re^n$ to a cell interpolating between the $2^n$ gradient vectors at the corners of the cell. In a typical application, the noise function is sampled frequently and so its efficiency is paramount. The standard implementation of Perlin's noise function uses a pair of precomputed arrays to allow fast calculation of the noise function. One of these arrays is an array of 256 random gradient vectors in $\Re^n$; the second is a random permutation array of byte-sized indices. The permutation array is used to map cell corners to indices in the gradient-vector array. In a language like SML, the performance of the noise function can be significantly reduced by the overhead of array bounds

```
abstype noise = N of {
      perm : Word8Array.array,
      grad : real array
    }
  in
  fun mkNoise () = let
        val perm = Word8Array.array(256, 0w0)
        val grad = Array(256, 0.0)
        ...
        in
          N{perm = perm, grad = grad}
        end
  fun noise (N{perm, grad}, x) = let
        val t = x + 512.0
        val b0 = (floor t) mod 256
        val b1 = (b0 + 1) mod 256
        val r0 = t - Real.realFloor t
        val r1 = r0 - 1.0
        val sX = sCurve r0
        fun get i =
              Array.sub(grad,
                Word8.toInt(
                  Word8Array.sub(perm, i)))
        val u = r0 * get b0
        val v = r1 * get b1
        in
          lerp (sx, u, v)
        end
  end
```

**Figure 7.** 1D Perlin Noise

checking,[9] so array-bounds-check elimination is an important optimization for this application. To make this example more concrete, Figure 7 sketches the code for 1D Perlin noise. The key point about this code is that the array subscript operations used to compute `u` and `v` can be statically eliminated as long as we know that the `perm` and `grad` arrays have 256 elements. Using our type-sensitive CFA, we can map the arrays used in `noise` back to the allocation sites in `mkNoise` and thus enable array-bounds-check elimination.

## 8. Related work

The application of control-flow analysis for higher-order functional languages dates back to Shivers' seminal work on control-flow analysis for SCHEME [Shi88, Shi91]. Many variations of this approach have been published including Serrano's algorithm on which we base the presentation in this paper [Ser95].

There is a significant body of work that falls into the intersection of type systems and program analysis. Some researchers have used control-flow analysis to compute type information for untyped languages [Shi91], while others have used type systems for program analysis [Pal01, Jen02].

Perhaps the most closely related work has been on using type information to guide analyses. For example, Jagannathan *et al.* devised a flow analysis for a typed intermediate language as one might find in an ML compiler. Their analysis uses type information to control polyvariance in the analysis and they prove that the analysis respects the type system [WJ98]. Saha *et al.* used type information to improve the performance of a demand-driven implementation of CFA [SHO98] in the SML/NJ compiler. Lastly, Diwan *et al.* used type information to improve alias analysis for MODULA-3 programs [DMM98]. We are not aware of any existing algorithms that use type abstraction to track values leaving and re-entering the unit of analysis as we do.

[9] For example, in the two-dimensional case, the noise function does 10 array subscript operations.

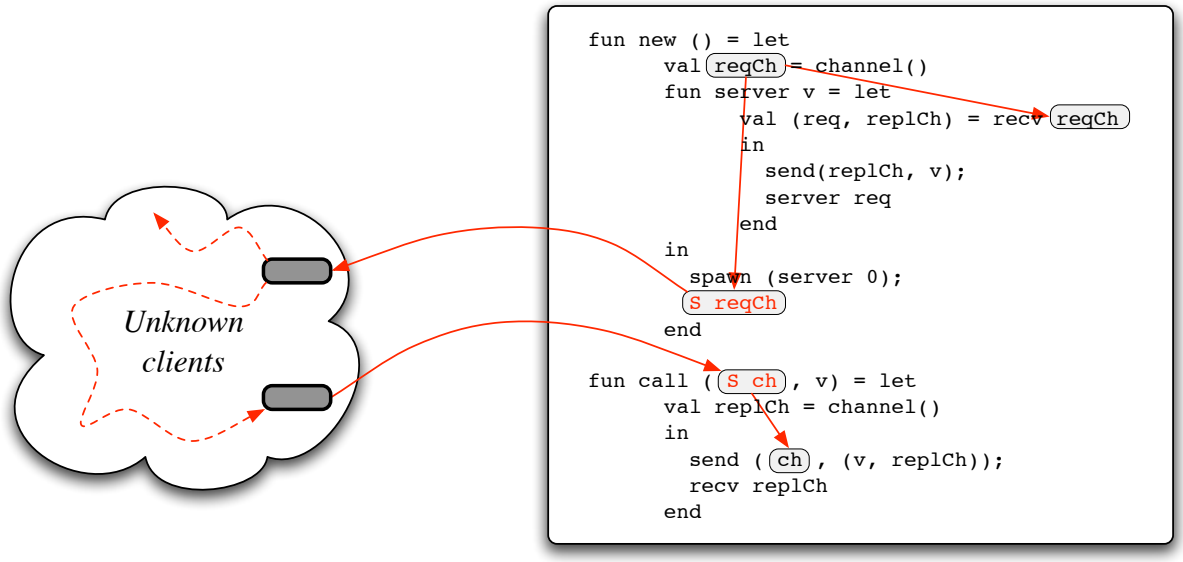


**Figure 6.** Data-flow of the server's request channel

## 9. Conclusion

We have presented an extension of control-flow analysis for ML that exploits type abstraction present in the source program to improve the quality of the analysis. Our technique is based on the combination of a more refined representation of unknown values and computing an approximation of abstract values that escape into the wild. We argued that our analysis is safe and faithful to the language's type system, and we gave two examples of optimizations that require the information provided by our analysis.

The key idea of tracking abstract values that escape could also be applied to other analysis frameworks, such as first-order data-flow analysis or more aggressive control-flow analyses (*e.g.*, 1-CFA [Shi91] or $\Delta$-CFA [MS06]). We presented our analysis for a monomorphic subset of SML This analysis is being used as part of an optimizer for CML and it will need to be extended to the full SML language. We described some ideas for how to extend the analysis, but more research is needed to fully deal with opaque signature ascription and polymorphism.

## References


[AD98] Ashley, J. M. and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, **20**(4), July 1998, pp. 845–868.

[App98] Appel, A. An alternate graph representation for the tiger compiler. `http://www.cs.princeton.edu/~appel/modern/ml/altgraph.html`, May 1998.

[CC77] Cousot, P. and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, January 1977. ACM, pp. 238–252.

[CDL99] Chilimbi, T. M., B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, New York, NY, May 1999. ACM, pp. 13–24.

[DMM98] Diwan, A., K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, New York, NY, June 1998. ACM, pp. 106–117.

[GMZ00] Grossman, D., G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, **22**(6), 2000, pp. 1037–1080.

[Jen02] Jensen, T. Types in program analysis. In *The essence of computation: complexity, analysis, transformation*, vol. 2566 of *Lecture Notes in Computer Science*, pp. 204–222. Springer-Verlag, New York, NY, 2002.

[Ler92] Leroy, X. Unboxed objects and polymorphic typing. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, New York, NY, January 1992. ACM, pp. 177–188.

[MS06] Might, M. and O. Shivers. Environment analysis via $\Delta$CFA. In *Conference Record of the 33rd Annual ACM Symposium on Principles of Programming Languages*, New York, NY, January 2006. ACM, pp. 127–140.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[NNH99] Nielson, F., H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, New York, NY, 1999.

[Pal01] Palsberg, J. Type-based analysis and applications. In *PASTE'01*, June 2001, pp. 20–27.

[Per02] Perlin, K. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, New York, NY, 2002. ACM, pp. 681–682.

[PH89] Perlin, K. and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, New York, NY, 1989. ACM, pp. 253–262.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, New York, NY, June 1991. ACM, pp. 293–305.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[RX06] Reppy, J. and Y. Xiao. Specialization of CML message-passing primitives. Submitted for publication., 2006.

[Ser95] Serrano, M. Control flow analysis: a functional languages compilation paradigm. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied Computing*, New York, NY, 1995. ACM, pp. 118–122.

[Shi88] Shivers, O. Control flow analysis in scheme. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, New York, NY, June 1988. ACM, pp. 164–174.

[Shi91] Shivers, O. *Control-flow analysis of higher-order languages*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991.

[SHO98] Saha, B., N. Heintze, and D. Oliva. Subtransitive CFA using types. *Technical Report YALEU/DCS/TR-1166*, Yale University, Department of Computer Science, New Haven, CT, October 1998.

[WJ98] Wright, A. K. and S. Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, **20**(1), 1998, pp. 166–207.

[Xia05] Xiao, Y. Toward optimization of Concurrent ML. Master's dissertation, University of Chicago, December 2005.