Type-sensitive control-flow analysis

John Reppy
Department of Computer Science
University of Chicago

jhr@cs.uchicago.edu

July, 18 2005

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 modular 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 a specific focus on optimizing Concurrent ML programs.

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.

[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].

In this paper, we present a modification of 0-CFA, a particular class of CFA [Shi88, Ser95]. Our modification exploits type abstraction to improve the accuracy of analysis (and thus enable more optimizations), without requiring whole-program analysis.[2] While our approach is presented in the context of a specific analysis, we believe that it is a general technique that can integrated into other flavors of flow-analysis for higher-order languages.

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. 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 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 require significantly more implementation complexity.

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. In Section 4, we address correctness issues with a focus on the type correctness of the analysis. We present our algorithm using a stripped down version of Core SML. In Section 5, 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 6. We review related work in Section 7 and conclude in Section 8.

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 motive our improvements.

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).

[2] Our technique does not provide an advantage of quality over whole-program CFA, but whole program analysis is often not a practical choice, since it interferes with separate compilation and does not scale well to large programs.

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

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:

```
abstype t = C of int list
  with
    fun new x = C[x]
    fun pick (C(x::_)) = x
  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 reflect 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** declaration to define abstractions in leu of modules. We further simplify this declaration to only have a single data constructor. Figure 1 gives the abstract syntax for this simple language. The top-level **abstype** declaration 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

[4]We discuss handling polymorphic types and user-defined type constructors in Section 5.

$$
\begin{array}{rcl}
d & ::= & \textbf{abstype}\, T = C\, \textbf{of}\, \tau\, \textbf{with}\, \mathit{fb}_1 \cdots \mathit{fb}_n\, \textbf{end} \\
\\
\mathit{fb} & ::= & \textbf{fun}\, f\,(x) = e \\
\\
e & ::= & x \\
 & | & \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \\
 & | & \textbf{fun}\, f\,(x) = e_1\, \textbf{in}\, e_2 \\
 & | & e_1\, e_2 \\
 & | & C\, e \\
 & | & \textbf{let}\, C\, x = e_1\, \textbf{in}\, e_2 \\
 & | & \langle e_1, \ldots, e_n \rangle \\
 & | & \#i\, e \\
\\
\tau & ::= & \mathrm{T} \\
 & | & \tau_1 \rightarrow \tau_2 \\
 & | & \tau_1 \times \cdots \times \tau_n
\end{array}
$$

Figure 1: A simple language

a collection of functions (fb_i). Outside the **abstype** declaration, 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 tuple construction and projection. Types include abstract types (T), function types, and tuple types. Abstract types are either predefined types (*e.g.*, `unit`, `int`, `bool`, *etc.*) or are defined by an **abstype** declaration.

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^τ). One should think of this language as a compiler's intermediate representation following typechecking.

We use LVAR to denote the set of variables defined in the program, GVAR to denote variables defined elsewhere, and VAR = LVAR$\cup$GVAR for all variables defined or mentioned in the program. We denote the known function identifiers by FUNID $\subset$ LVAR (*i.e.*, those variables that are defined by function bindings). The set ABSTY is the set of abstract type names and DATACON is the set of data constructors.

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

3.2 Abstract values

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

$$\begin{array}{rcl} v & ::= & \bot \\ & | & C\,v \\ & | & \langle v_1, \ldots, v_n \rangle \\ & | & F \\ & | & \widehat{T} \\ & | & \widehat{\tau_1 \to \tau_2} \\ & | & \top \end{array}$$

where $C \in \textsc{DataCon}$, $F \in 2^{\textsc{FunId}}$, and $T \in \textsc{Absty}$. We use $\bot$ to denote undefined or not yet computed values, $C\,v$ for an approximate value constructed by applying C to v, $\langle v_1, \ldots, v_n \rangle$ for an approximate tuple, 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 4 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 τ (where τ is either a function or abstract type). The auxiliary function $\mathcal{U} : \textsc{Type} \to \textsc{Value}$ maps types to their corresponding top value:

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

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:

$$\begin{array}{rcc} \bot \vee v & = & v \\ v \vee \bot & = & v \\ C\,v_1 \vee C\,v_2 & = & C(v_1 \vee v_2) \\ \langle v_1, \ldots, v_n \rangle \vee \langle v'_1, \ldots, v'_n \rangle & = & \langle v_1 \vee v'_1, \ldots, v_n \vee v'_n \rangle \\ F \vee F' & = & F \cup F' \\ \top \vee v & = & \top \\ v \vee \top & = & \top \\ \widehat{\tau} \vee v & = & \widehat{\tau} \\ v \vee \widehat{\tau} & = & \widehat{\tau} \end{array}$$

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 4 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:

$$\begin{array}{rcl}
\lceil \bot \rceil_{\mathbf{C}} & = & \bot \\
\lceil C^{\tau \to T}\, v \rceil_{\mathbf{C}} & = & \begin{cases} \top & \text{if } C \in \mathbf{C} \\ C(\lceil v \rceil_{\mathbf{C} \cup \{C\}}) & \text{if } C \notin \mathbf{C} \end{cases} \\
\lceil \langle v_1, \ldots, v_n \rangle \rceil_{\mathbf{C}} & = & \langle \lceil v_1 \rceil_{\mathbf{C}}, \ldots, \lceil v_n \rceil_{\mathbf{C}} \rangle \\
\lceil F \rceil_{\mathbf{C}} & = & F \\
\lceil \widehat{\tau} \rceil_{\mathbf{C}} & = & \widehat{\tau}
\end{array}$$

where $\mathbf{C} \subset$ DATACON is a set of constructors. We write $\lceil v \rceil$ for $\lceil v \rceil_{\emptyset}$. We use $\top$ to cutoff the expansion of approximate values instead of $\widehat{T}$ 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 uses a triple of approximations: $\mathcal{A} = (\mathcal{V}, \mathcal{R}, \mathcal{T})$, where

$\mathcal{V}$	$\in$	VAR $\to$ VALUE	variable approximation
$\mathcal{R}$	$\in$	FUNID $\to$ VALUE	function-result approximation
$\mathcal{T}$	$\in$	ABSTY $\to$ 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 leu 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 an object-language syntactic form e and $\mathcal{V}[x \mapsto v]$ to denote the functional update of an approximation (likewise for $\mathcal{R}$ and TA).

Our unit of analysis is the abstype declaration. Our algorithm analyses the function definitions in the declaration repeatedly until a fixed-point is reached. The initial approximation 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} ∪ {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 declaration by "applying" the function to the top value of the function's argument type. The result is then recorded as escaping.

```
fun cfaFB (A, ⟦fun f (x^τ) = e⟧) = let
      val (A, v) = applyFun ({}, A, f, U(τ))
      in
        escape ({}, A, v)
      end
```

The `applyFun` function analyses the application of a known function f to an approximate value 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 the function `bindingOf` that maps known function names to their bindings in the source.

```
fun applyFun (M, A as (V, R, T), f, v) =
      if f ∈ M
        then (A, R(f))
        else let
          val [[fun f (x) = e]] = bindingOf (f)
          val V = V[x ↦ ⌈V(x) ∨ v⌉]
          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
```

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 them to the appropriate top value; and if it is a tuple, we record that its subcomponents are escaping. The `escape` function also takes the set of currently active functions as its first argument.

```
fun escape (_, (V, R, T), C v) =
      (V, R, T[T ↦ ⌈T(T) ∨ C v⌉])
  | escape (M, A, F) = let
      fun esc (f^τ1 → τ2, A) = let
            val (A, v) = applyFun(M, A, f, U(τ1))
            in A end
      in
        fold esc A F
      end
  | escape (M, A, ⟨v1, ..., vn⟩) = let
      val A = escape (M, A, v1)
      ...
      val A = escape (M, A, vn)
      in A end
  | escape (_, A, v) = 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 cfaExp (M, A as (V, R, T), [[x]]) =
      if x ∈ FunId then (A, {x}) else (A, V(x))
  | cfaExp (M, A, [[let x = e1 in e2]]) = let
      val ((V, R, T), v) = cfaExp (M, A, [[e1]])
      val V = V[x ↦ ⌈V(x) ∨ v⌉]
      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, v1) = cfaExp (M, A, [[e1]])
      val (A, v2) = cfaExp (M, A, [[e2]])
      in
        apply (M, A, v1, v2)
      end
  | cfaExp (M, A, [[C e]]) = let
      val (A, v) = cfaExp (M, A, [[e]])
      in
        (A, C v)
      end
  | cfaExp (M, A, [[let C x = e1 in e2]]) = let
      val ((V, R, T), v) = cfaExp (M, A, [[e1]])
      val V = decon (V, T, [[C x]], v)
      in
        cfaExp (M, (V, R, T), [[e2]])
      end
  | cfaExp (M, A, [[⟨e1, ..., en⟩]]) = let
      val (A, v1) = cfaExp (M, A, [[e1]])
      ...
      val (A, vn) = cfaExp (M, A, [[en]])
      in
        (A, ⟨v1, ..., vn⟩)
      end
  | cfaExp (M, A, [[#i e]]) = let
       val (A, ⟨v1, ..., vn⟩) = cfaExp (M, A, [[e]])
       in
         (A, vi)
       end
```

Figure 2: CFA for expressions

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

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

4 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. We believe that the proof of this property for other versions of CFA carry over directly to our algorithm. Likewise, the question of whether the algorithm terminates has been addressed by Serrano [Ser95] (the bounding of the sizes of abstract values is crucial to the termination of the fixed-point iteration). In this section, we focus on a third correctness issue, which is the question of type correctness; *i.e.*, we show that the approximation that our algorithm computes is type correct.

$$\frac{\vdash\ \mathit{fb}_1\ :\ \mathbf{Ok} \quad \cdots \quad \vdash\ \mathit{fb}_n\ :\ \mathbf{Ok}}{\vdash\ \mathbf{abstype}\ T\ =\ C^{\tau\to T}\ \mathbf{of}\ \tau\ \mathbf{with}\ \mathit{fb}_1\ \cdots\ \mathit{fb}_n\ \mathbf{end}\ :\ \mathbf{Ok}}$$

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

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

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

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

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

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

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

$$\frac{\vdash\ e_1\ :\ \tau_1 \quad \cdots \quad \vdash\ e_n\ :\ \tau_n}{\vdash\ \langle e_1,\ \ldots,\ e_n\rangle\ :\ \tau_1\times\cdots\times\tau_n}$$

$$\frac{\vdash\ e\ :\ \tau_1\times\cdots\times\tau_n \qquad 1\le i\le n}{\vdash\ \#i\,e\ :\ \tau_i}$$

Figure 3: Typing rules for declarations, function bindings, and expressions

4.1 Code typing rules

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

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

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

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

$$\frac{\vdash\ v_1\ :\ \tau_1 \quad \cdots \quad \vdash\ v_n\ :\ \tau_n}{\vdash\ \langle v_1, \ldots, v_n\rangle\ :\ \tau_1 \times \cdots \times \tau_n}$$

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

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

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

Figure 4: Typing rules for approximate values

4.2 Value typing rules

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.

4.3 Type correctness of the analysis

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^τ defined or mentioned in the 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* $\vdash\ v_1\ :\ \tau$ *and* $\vdash\ v_2\ :\ \tau$, *then* $\vdash\ v_1 \vee v_2\ :\ \tau$.

> ***Proof:*** *The proof two parts. First, we can prove that if* $\vdash\ v_1\ :\ \tau$ *and* $\vdash\ v_2\ :\ \tau$, *then* $v_1 \vee v_2$ *is defined (recall that* $\vee$ *is partial) by case analysis of the shape of* τ *and the*

typing rules. Then, a simple inductive argument shows that $\vdash v_1 \vee v_2 : \tau$ □

Theorem 2 *Given an expression* e*, with* $\vdash e : \tau$*, and a type consistent approximation triple* $\mathcal{A}$*,* $\mathcal{V}$*, and* $\mathcal{T}$*, if* `cfaExp`$(\{\}, \mathcal{A}, e)$ *returns the result* $(\mathcal{A}', v)$*, then* $\mathcal{A}'$ *is type consistent and* $\vdash v : \tau$*.*

> ***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 $\mathcal{V}$ and $\mathcal{T}$ in `cfa` are type consistent, the above theorem guarantees that the resulting approximation will be type consistent.

5 Extensions

We presented our algorithm as a modification of 0-CFA, but we see no reason why our techniques will not extend to similar program analyses (*e.g.*, 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 to other ML features.

5.1 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. We propose 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 $C(\widehat{\tau})$, instead of the $\perp$. This initialization reflects the fact that clients of the module can construct values of type T using the constructor C.

Another 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 member : (set * int) -> bool
    ...
  end = struct
    type set = int list
    fun singleton x = [x]
    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. We believe that we can use the signature as a guide to rewrite the module's code by adding annotations to mark the abstractions. 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.

5.2 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.

5.3 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 some of the aspects of shape analysis [NNH99, Section 2.6] and one might want a cutoff on recursive values that is more generous.

```
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

6 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 applications where we believe our techniques are necessary to achieve the desired optimizations. We discuss these applications below.

6.1 Optimizing CML

The algorithm presented in this paper is part of an effort to develop analysis and optimization techniques for CML-style concurrency features. For example, consider the simple service implemented in Figure 5. The `new` function creates a new service, which is represented by an abstract type, and the `call` function allows clients to request the service. Our goal is to develop analyses 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. These properties allow the optimizer to replace the general channel operations with more

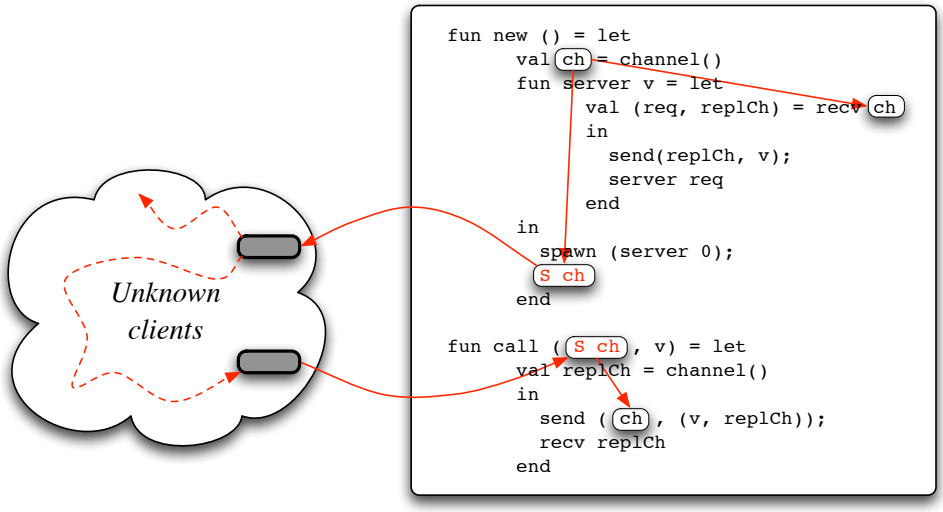

Figure 6: Data-flow of the server's request channel

specialized ones.

Figure 6 illustrates the flow of the server's request channel. Note that while the unknown clients of the service may store the a `serv` value in data structures, *etc.*., they may not access the internal representation and perform operations directly on the request channel. The analysis presented in this paper can detect this flow pattern (when extended for concurrency operations).

6.2 Representation optimization

There are a number of techniques for optimizing data representations that can be applied when one knows all of the sites where the data structure is created and accessed. For example, data structures might be flattened, which reduces space and improves access time, and fields might be packed (*i.e.*, 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]. Our analysis can provide the information to enable these transformations.

7 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]. It is likely that a type-bvased CFA could be extended to exploit type abstraction (presumably via the addition of singleton types). We chose to base our analysis on an abstract-interpretation style algorithm because we find it more intuitive.

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 [JWW97]. 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 textscModula-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.

8 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 showed that our analysis respects types (*i.e.*, computes a type-correct approximation) and we discussed several applications with a special focus on optimizing CML programs.

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.

[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.

[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.

[JWW97] Jagannathan, S., S. Weeks, and A. K. Wright. Type-directed flow analysis for typed intermediate languages. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, vol. 1302 of *Lecture Notes in Computer Science*, New York, NY, 1997. Springer-Verlag, pp. 232–249.

[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.

[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.

[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.