Modular Cloning*

Karl-Filip Faxén
Swedish Institute of Computer Science, Kista
kff@sics.se

April 10, 2008

SICS Technical Report T2008:07, ISSN 1100-3154

ABSTRACT

In this paper we deal with the problem of making context dependent interprocedural optimizations (where the legality of optimizing a function depends on properties of the callers of the function) effective and compatible with (a form of) separate compilation. We improve effectiveness by cloning, generating several versions of a single function optimized for different call sites.

We attack the separate compilation problem, that code can not be generated until all calls of a function are known, by splitting the compilation process into two phases. The first phase analyses the modules one at a time in bottom-up dependency order (`main` is processed last) and produces code in an intermediate language where the constructs targeted by the optimization are annotated to control the application of the optimization. In cases where the legality of an optimization depends on properties of the callers of the function, these annotations can take the form of annotation variables which become extra formal parameters. The second phase traverses the modules in top-down dependency order, removing all of these extra parameters by specialization.

We illustrate our approach with an integrated programming analysis and transformation system featuring a context sensitive type based analysis, cloning with sharing of identical clones and a modular implementation allowing for the compilation of large programs. The system implements cheap eagerness and redundant `eval` elimination for a lazy functional language.

Keywords: Modular compilation, cheap eagerness, cloning, static analysis, type inference, functional programming, optimization, program transformation

1. INTRODUCTION

*Based on an earlier unpublished draft.

Many useful compiler optimizations depend on properties of the calls to the optimized procedure or function. In a lazy functional language, for instance, such a property may be that a function is only called with evaluated arguments, in which case testing for unevaluated arguments becomes redundant and the opportunities for applying the cheap eagerness optimization [13] increase. While these transformations are specific to lazy languages, compilers for any language may benefit from knowing for instance that certain arguments are constants, another call site dependent property.

Techniques like these give significant speedups, in particular for lazy functional languages [13], but they are limited in their effectiveness by the need to be safe with respect to every call to the optimized function. For functions used at many call sites, this means that if just one call site invalidates an optimization, the performance of every call to the function will be hurt. For large programs using many library functions, this problem is severe.

To eliminate this "crosstalk" several versions of each function can be generated, each one tailored to a subset of the calls to the function. This transformation is called *cloning* [6] and has been used in imperative languages [17, 1] as well as in object oriented languages[1] [3, 10, 20] and it has also been shown to provide substantial benefits to functional programs [14].

A second problem with context dependent optimization is that separate compilation becomes very difficult since the code generated for a function `f` depends on properties of the callers of `f`, which may reside in modules importing the module defining `f`. At the same time, analysing the importing modules may need analysis information regarding `f`, so it is difficult to decide which module should be compiled first. If cloning is used, it is not known which clones are needed until importing modules are compiled. Hence systems which use context dependent optimization tend to do whole program optimization [20].

The solution to this dilemma can be guessed from careful reading of the previous paragraph: For analysis, information about the *imported* modules is needed, while for code generation it is the *importing* modules that must be processed first. Thus we separate analysis and code generation

[1]In OO languages, what we call cloning in this paper is often termed *customization* or *specialization*.

and do the analysis bottom up and the code generation top down. The key to modular cloning is an intermediate language which can glue these two passes together.

An immediate consequence of this approach is that we do not handle mutually recursive modules directly. In practice, mutually recursive modules can be merged by the compiler front-end.

We see modular cloning as a generic approach to making context dependent analysis *effective* (by cloning) and *efficient* (by making it modular). At the core of this approach is

- an intermediate language where each unit of cloning (here function) is parameterized over those parts of the context that affect its optimization,
- a modular analyser capable of reasoning about the unknown context and finding how it affects optimization (we use a type based analyser with `let`-polymorphism), producing code in the intermediate language, and
- a specializer that generates exactly those clones that are needed, one module at a time, translating from the intermediate language.

We have formulated several optimizations (cheap eagerness, redundant `eval` elimination, update avoidance and representation selection) in this way; we will use a combination of cheap eagerness and redundant `eval` elimination as a case study in this paper. We also think that modular cloning is applicable to other types of languages (imperative, object oriented, ...).

We believe that the main contribution of the paper is the invention of the parameterized intermediate language and its relation to the analyser since it clarifies exactly what part of the optimizations are context dependent. This is in contrast to most work on analysis-based optimization where the focus is very much on the analysis and the transformation part is playing a secondary role.

Modular cloning rests on modular analysis. In this paper we use a type system with `let`-polymorphism, but there is quite a lot of freedom in the choice of analysis technology. First, there are other type systems using combinations of subtyping, more powerful kinds of polymorphism and intersection types. Soft typing can be used if the source language is not statically typed or to allow the use of an impredicative type system. Second, analysers based on constraint solving can also be used, as can more *ad hoc* data flow techniqes like relevant context inference [4].

The example analysis presented is the first type based cheap eagerness analysis, but for simplicity of presentation, it is not as precise as our earlier non-type based work [13], the main difference being that the present analyser considers all function calls expensive.

The rest of the paper is organized as follows: The rest of this section gives an informal overview of modular cloning and the example application. Section 2 presents the parameterized intermediate language and section 3 gives a specification of the analysis in terms of an inference system. Both of these sections essentially deal with the correctness of the analysis. Section 4 gives an implementation of the analysis in the form of an inference algorithm and section 5 presents the specialization (cloning) algorithm. In section 6 we discuss how we can avoid having to recompile the entire program after a a subset of the source modules have been updated. Section 7 discusses related work and section 8 concludes.

```
module libfun = λf. λn. (eval f) (thunk (eval n)+1)
module appfun = λg. λm. libfun (eval g) 3 + libfun (eval g) m
module main = appfun (λx. eval x) 3
            + appfun (λx. 1) (thunk 1/0)
```

Figure 1: A simple program

1.1 The example analysis

Consider the example program given in figure 1. It consists of three modules, each exporting a single binding. The value `main` is an integer while the two other modules export functions; `appfun`, whose name is meant to suggest that it is part of the application and `libfun` which might be a library function.

The functions are written in a functional language where the order of evaluation has been made explicit. Delayed (lazy) evaluation is expressed using the `thunk` construct; `thunk` e constructs a representation for the unevaluated expression e (typically a small record containing a code pointer and the values of the free variables of e). The `eval` operation resumes the delayed computation, yielding an evaluated result (typically by an indirect function call to the code pointer with the address of the thunk as argument). If the argument of the `eval` is not a thunk, the `eval` simply returns it. This language is suitable as an intermediate language in a compiler for lazy functional languages such as Haskell [19].

Context independent optimization

We assume that a simple strictness analysis has already been applied. For instance, in `appfun` the compiler has inserted an `eval` around the first argument to `libfun` in both applications. It can do this since `libfun` will certainly evaluate it anyway. This use of strictness analysis is *context independent* (bottom-up); the caller (`appfun`) is adapted to the callee (`libfun`) and can be implemented by compiling the `libfun` module before the `appfun` module and recording the strictness of `libfun` in its interface file. All major Haskell compilers implement this optimization.

Other context independent techniques include *inlining* [1], where the caller is adapted to the callee by replacing the call with a copy of the code of the callee, and some forms of *interprocedural register allocation*, where the caller can keep values in registers across a call if the callee is known not to overwrite these registers.

Context dependent optimization

```
module libfun = {u1, u2} λf. λn. (eval{u1} f)
                                 (thunk{u2} (eval{u2} n)+1)

module appfun = {u3, u4} λg. λm. libfun{F, F} (eval{u3} g) 3 +
                                 libfun{F, u4} (eval{u3} g) m

module main = appfun{F, F} (λx. eval{F} x) 3
            + appfun{F, T} (λx. 1) (thunk{T} 1/0)
```

Figure 2: The example translated into the intermediate language

The `thunk` in `libfun` can not be eliminated by strictness analysis since we do not know if the first argument to `libfun` will always be a strict function. This is clearly a *context dependent* (top-down) property.

The most well known example of context dependent optimization is probably *interprocedural constant propagation* [17] where a formal parameter can be replaced by a constant if the procedure is always called with that constant as argument. This can be very important if that parameter controls data access patterns or loop structures, making paralellization and vectorization more effective.

Another example is *pointer analysis* [9], where a superset of the possible targets for each pointer dereference in a program is computed. This gives *e.g.* alias information, which can be used to improve register allocation and code scheduling.

Cheap eagerness

Strictness analysis finds cases where a value can be evaluated early because it is certain to be needed later, even if this early evaluation might diverge or raise a run-time error. In contrast, cheap eagerness [18, 13] evaluates an expression early because its evaluation is guaranteed not to diverge or raise a run-time error, even if it is not certain that the value of the expression is really needed. This is beneficial if evaluating the expression is cheaper than building the thunk or if the thunk is likely to have been evaluated anyway (this is more likely than not). There are also secondary effects since `eval` operations can be removed and unboxed data representations used more often.

Thus the task of a cheap eagerness analyser is to find `thunk`s whose bodies are cheap and safe. We will consider variable references, constants, some operators (*e.g.* addition but not division), thunks and lambda abstractions as cheap (the latter two since they only allocate something in the heap). Function calls,[2] some operators and `eval`s are unsafe. Note that if we know that the argument of a certain `eval` will never be a thunk, that `eval` is *redundant* and can be eliminated. If the `eval` was part of the body of a `thunk`, this `thunk` might now be cheap.

This is the core analysis problem we will use as example of modular cloning in this paper. Note that cheap eagerness and redundant `eval` elimination are mutually dependent and must be performed at the same time. It takes only one glance at `libfun` to realize that this is clearly a context dependent problem: The `eval` of `n`, and then also the `thunk`, can be eliminated if the value of `n` (the second argument to `libfun`) is not a thunk.

[2]A sophisticated cheapness analyser can determine that some function calls are cheap, but in this paper we will, for simplicity, consider all calls expensive.

Cloning

The trouble is that `libfun` *will* be called with a thunk as second argument. The second call to `appfun` in `main` passes a thunk and the second call to `libfun` in `appfun` passes that thunk through. This is a pity since the first call in `appfun` passes an evaluated constant; adding insult to injury, the transformation interacts badly with separate compilation since code can not be generated for `libfun` until the whole program has been analysed.

We will deal with the first problem by cloning, that is, generating several different versions of the same function. Thus we will have one version of `appfun` without any `thunk`s or `eval`s and one with only the `eval` of `f` removed.

In the limit, each call site invokes its own tailor made clone but in general, different call sites are similar enough that fewer clones than call sites are needed. For instance, in constant propagation all call sites that pass non-constant arguments may invoke the same clone and so can call sites that pass the same constant. The increase in code size entailed by cloning is in most cases offset by the simplification of the resulting code and the net increase in object code size is typically modest. Nevertheless, cloning often increases the number of misses in the instruction cache [14], a cost that must be weighed against the benefits imparted by the more agressive optimization. We have seen reductions in execution time of 11–29% for some small ($\leq$ 700 lines) lazy functional programs.

Modular cloning

We address the second problem (that code can not be generated until the whole program is analysed) by parameterizing the code with respect to the context dependent optimization decisions. We do this by annotating the `thunk`s and `eval`s in the program with either `T`, meaning that the construct must stay, `F`, meaning that the construct can safely be optimized away, or a variable, meaning that it depends. For each of the functions we will add the new annotation variables as a kind of extra formal parameters. This means that we must also add corresponding extra arguments to every call to `libfun` and `appfun`. The result of this transformation is shown in figure 2.

For `libfun`, note that the `eval` of `f` can be omitted if `f` is not a thunk; since this depends on the caller of `libfun`, we annotate the `eval` with the variable u_1. The `thunk` and the other `eval` depend on the evaluatedness of `n`, the second argument, and can be annotated with the same variable u_2. This analysis is context independent; the other modules need not be consulted. Apart from the transformed code, we also note an analysis result (not shown in the figure) which says that

- if the first argument in a call to `libfun` is evaluated, we may instantiate the first cloning parameter to `F`(and

```
module libfun_FF = λf. λn. f (n+1)
       libfun_FT = λf. λn. f (thunk (eval n)+1)

module appfun_FF = λg. λm. libfun_FF g 3 + libfun_FF g m
       appfun_FT = λg. λm. libfun_FF g 3 + libfun_FT g m

module main = appfun_FF (λx. x) 3
            + appfun_FT (λx. 1) (thunk 1/0)
```

Figure 3: The example specialized

similarly for the second argument and second cloning parameter), and

- the first argument is a function that will be applied to something that might be a thunk if the second argument is a thunk.

For `appfun`, we have the `eval`s of `g` and the two calls to the cloned `libfun` to annotate. In the first call, we can place ground annotations; we know that both arguments are evaluated. In the second call, the second parameter is context dependent. Thus the optimization of a construct might depend on some function several layers up in the call graph. The analysis result is similar to that for `libfun`.

Finally, `main` is analysed. We can see that in both calls, the first argument is evaluated. The `eval` in the identity function is in fact redundant since the second argument in this call (`3`) is evaluated and it is this value that (in `libfun`) will be passed to the identity function. In order to figure this out, the analyser has to be able to track the higher order control flow of the program, something that our type-based approach handles easily.

Specialization

At this point, it would be possible to generate code for the program that explicitly manipulates the annotations at runtime. That would however have the drawback that we replace the overhead of (some of) the lazy evaluation with the overhead of the extra parameterization. Instead, we will eliminate it by specialization. While the analysis and generation of the intermediate code proceeded bottom-up, our specialization will proceed top-down. Looking at figure 2 it is easy to see why: There are no annotation variables in `main`. Any modular cloning analysis must ensure this by arranging for the context of `main` to be known.

2. THE LANGUAGES

There are in general three languages involved in an optimizer based on modular cloning:

- The input language is a conventional language without annotations or cloning constructs (in the example analysis used in this paper, we call it λ^{in}). It might either be a source language or, as in this case, an intermediate language produced by the front-end and already optimized using local and context independent techniques.

$$
\begin{array}{lll}
ms \in \text{Modules} & \to m_1; \ldots; m_n & \\
m \in \text{Module} & \to \texttt{module}\ x = e & \\
e \in \text{Exp} & \to x \mid e_1\, e_2 \mid \lambda x.\, e \mid op\ e_1 \ldots e_r & \\
 & \mid \texttt{let}\ x = e\ \texttt{in}\ e' & \\
 & \mid \texttt{thunk}\{a\}\, e \mid \texttt{eval}\{a\}\ e & \\
 & \mid \{\vec{z}\}\, b \mid x\{\vec{a}\} & \\
b \in \text{Build} & \to op \mid \lambda x.\, e \mid \texttt{thunk}\{\mathsf{T}\}\ e \mid \{\vec{z}\}\, b &
\end{array}
$$

$$
\begin{array}{lll}
a \in \text{Bool Exp} & \to z \mid \mathsf{F} \mid \mathsf{T} & \\
\underline{a} \in \text{Bool Val} & \to \mathsf{F} \mid \mathsf{T} &
\end{array}
$$

$x \in \text{Var}, \quad z \in \text{Bool Var}$

Figure 4: The syntax of λ^{mc}

- The intermediate language is the input language plus annotations and cloning abstractions and applications (λ^{mc} in this paper). There should be a trivial transformation $\mathcal{A}$ which maps the input language to the intermediate language by adding conservative ("don't optimize") annotations.
- The output language is the target language of the specializer (here called λ^{out}). It has only ground (variable free) annotations and lacks cloning abstractions and applications.

Since they are so similar, it is convenient to give only one semantics to these three languages. Thus we only give a semantics directly to the intermediate language (λ^{mc}) and define the semantics of a λ^{in} expression e as the semantics of its trivial translation into λ^{mc}, $\mathcal{A}(e)$. The output language λ^{out} is a subset of λ^{mc}, so there we use the semantics of λ^{mc} directly.

Figure 4 gives the syntax of λ^{mc}, which is a call-by-value functional intermediate language with explicit `thunk` and `eval` constructs for expressing lazy evaluation. Constants (nullary operators), lambda abstractions, cloning abstractions, and thunks are *buildable expressions* (in a real implementation, these will be implemented by building a new heap cell).

There is no built-in evaluation of thunks in the language; if the value of a variable is needed in evaluated form, an explicit `eval` must be used unless the variable is known to have only evaluated values. In those situations where a thunk is acceptable, an evaluated value is however also acceptable.

A module is a binding prefixed by the keyword `module`, and a program is a sequence of modules where later modules may depend on earlier ones (no mutual recursion). We will write $m; ms$ for the module m followed by the module sequence ms as well as $ms; m$ for the sequencs ms followed by the module m. We will use the name of the bound variable x as the name of the module `module` x `=` e and we will say that it exports x and imports $fv(e)$ (the free variables of e).

A technical requirement is that dead code elimination has been performed; in a `let`-expression `let` x `=` e `in` e', x must occur free in e', and similarly, each module must be used in a later one except for `main`.

$\boxed{\rho \vdash Pgm \Downarrow w}$

$$\frac{\rho, \mathsf{T} \vdash e \Downarrow w \qquad \rho[x \mapsto w] \vdash ms \Downarrow w'}{\rho \vdash \texttt{module}\ x = e;\ ms \Downarrow w'} \quad \text{module}$$

$$\rho \vdash \epsilon \Downarrow \rho(\texttt{main}) \quad \text{main}$$

$\boxed{\rho, \underline{a} \vdash e \Downarrow w}$

$$\rho, \underline{a} \vdash x \Downarrow \rho(x) \quad \text{var}$$

$$\frac{\rho, \mathsf{T} \vdash e_1 \Downarrow (\rho', \lambda x.\, e) \qquad \rho, \mathsf{T} \vdash e_2 \Downarrow w \qquad \rho'[x \mapsto w], \mathsf{T} \vdash e \Downarrow w'}{\rho, \mathsf{T} \vdash e_1\, e_2 \Downarrow w'} \quad \text{app}$$

$$\rho, \underline{a} \vdash \lambda x.\, e \Downarrow (\rho, \lambda x.\, e) \quad \text{abs}$$

$$\frac{\rho, \underline{a} \vdash e_1 \Downarrow w_1 \quad \ldots \quad \rho, \underline{a} \vdash e_r \Downarrow w_r \qquad [\![op]\!]\, \underline{a}\, w_1 \ldots w_r = w}{\rho, \underline{a} \vdash op\, e_1 \ldots e_r \Downarrow w} \quad \text{op}$$

$$\frac{\rho, \underline{a} \vdash e' \Downarrow w' \qquad \rho[x \mapsto w'], \underline{a} \vdash e \Downarrow w}{\rho, \underline{a} \vdash \texttt{let}\ x = e'\ \texttt{in}\ e \Downarrow w} \quad \text{let}$$

$$\frac{\rho(a) = \mathsf{T}}{\rho, \underline{a} \vdash \texttt{thunk}\{a\}\, e \Downarrow (\rho, \texttt{thunk}\{\mathsf{T}\}\, e)} \quad \text{thunk-i}$$

$$\frac{\rho(a) = \mathsf{F} \qquad \rho, \mathsf{F} \vdash e \Downarrow w}{\rho, \underline{a} \vdash \texttt{thunk}\{a\}\, e \Downarrow w} \quad \text{thunk-ii}$$

$$\frac{\rho, \mathsf{T} \vdash e \Downarrow (\rho', \texttt{thunk}\{a'\}\, e') \qquad \rho(a) = \mathsf{T} \qquad \rho', \mathsf{T} \vdash e' \Downarrow w}{\rho, \mathsf{T} \vdash \texttt{eval}\{a\}\, e \Downarrow w} \quad \text{eval-i}$$

$$\frac{\rho, \mathsf{F} \vdash e \Downarrow w \qquad w \text{ is a whnf closure}}{\rho, \underline{a} \vdash \texttt{eval}\{a\}\, e \Downarrow w} \quad \text{eval-ii}$$

$$\rho, \underline{a} \vdash \{\vec{z}\}\, b \Downarrow (\rho, \{\vec{z}\}\, b) \quad \text{clone}$$

$$\frac{\rho, \underline{a} \vdash e \Downarrow (\rho', \{\vec{z}\}\, b)}{\rho, \underline{a} \vdash e\{\vec{a}\} \Downarrow (\rho'[\vec{z} \mapsto \rho(\vec{a})], b)} \quad \text{inst}$$

Error rules

$$\rho, \mathsf{F} \vdash e_1\, e_2 \Downarrow \mathsf{error} \quad \text{app err}$$

$$\frac{\rho, \underline{a} \vdash e \Downarrow (\rho', \texttt{thunk}^{a'}\, e') \qquad \underline{a} = \mathsf{F} \text{ or } \rho(a) = \mathsf{F}}{\rho, \underline{a} \vdash \texttt{eval}\{a\}\, e \Downarrow \mathsf{error}} \quad \text{eval err}$$

$$\frac{\rho(x) = \mathsf{error} \text{ for some } x \in dom(\rho)}{\rho, \underline{a} \vdash e \Downarrow \mathsf{error}} \quad \text{env err}$$

$$\frac{\rho, \underline{a} \vdash e \Downarrow (\rho', e') \qquad e' \text{ is not a clone abs matching } \vec{a}}{\rho, \underline{a} \vdash e\{\vec{a}\} \Downarrow \mathsf{error}} \quad \text{inst err}$$

Figure 5: The semantics of λ^{mc}

We give λ^{mc} a big step operational semantics in Figure 5. The semantics allows us to prove judgements of the form $\rho, \underline{a} \vdash e \Downarrow w$ where ρ is a *value environment* mapping program variables to values and annotation varibles to boolean values ($\{\mathsf{F}, \mathsf{T}\}$), $\underline{a}$ is a boolean value called the *evaluation budget* that controls whether the expression is allowed to perform function calls and operations which might loop or raise exceptions ($\underline{a} = \mathsf{T}$) or if only cheap expressions are allowed ($\underline{a} = \mathsf{F}$), e is an expression and w is a *value*. A value w is either a closure (ρ, b), where ρ is a value environment and b is a buildable expression, or error. We extend environments to boolean expressions by $\rho(\mathsf{F}) = \mathsf{F}$ and $\rho(\mathsf{T}) = \mathsf{T}$. Since λ^{mc} is a call-by-value language, a thunk is a value like any other; we use the term *whnf closure* to refer to a non-thunk closure.

2.1 Annotations and cloning

Some constructs in λ^{mc} are annotated with information controlling their semantics. This information may be given in the form of annotation variables z which are bound in *cloning abstractions*. In this way, a single cloning abstraction may represent several versions of an expression, optimized for use in different contexts, as discussed in Section 1.1. Intermediate code can be generated before enough of the context is known to determine the legality of every optimization.

A `thunk` expression of the form $\texttt{thunk}\{a\}\ e$ has a boolean annotation a which controls whether the thunk is built ($a = \mathsf{T}$) or speculatively evaluated (the cheap eagerness transformation [13]). An expression of the form $\texttt{thunk}\{\mathsf{F}\}\ e$ can be translated to e.

An explicit evaluation operation of the form $\texttt{eval}\{a\}\ e$ is annotated with a boolean a which allows the test for a thunk closure to be omitted if $a = \mathsf{F}$. This optimization is only legal if the argument e never evaluates to a thunk. An expression of the form $\texttt{eval}\{\mathsf{F}\}\ e$ can be translated to e.

2.2 Annotations in the semantics

The semantics of λ^{mc} checks that the program is annotated correctly; if a violation is detected, the value error becomes derivable. This is formalized in the last four rules in figure 5. The first one, [app err], deals with an attempt to evaluate a function application in a speculative context (the only rule with a premise of the form $\rho, \mathsf{F} \vdash e \Downarrow w$ is [thunk-ii]). The second rule, [eval err], catches two errors: Finding a thunk at an `eval` marked as redundant ($\rho(a) = \mathsf{F}$) or in a speculative context ($\underline{a} = \mathsf{F}$). The third rule, [env err], indicates that if the environment maps some variable to error, then the result is also error. Finally, the [inst err] rule covers the case of malformed cloning applications.

The error rules make the semantics of λ^{mc} nondeterministic; sometimes, both error and an ordinary result can be derived. In these cases the program is still considered incorrect.

The reason for including the error checking rules is to be able to say that if evaluation of an expression cannot lead to error, the semantics of the expression is, in an abstract sense, the same as if no optimization had been applied to the expression.

$\sigma \in$ Scheme $\rightarrow \forall \vec{\beta}, \vec{\alpha}.\tau \mid P$
$\tau \in$ Annotated type $\rightarrow (\eta, \nu)$
$\nu \in$ Ordinary type $\rightarrow \alpha \mid \tau_1 \rightarrow \tau_2 \mid U \mid \vec{\eta} \Rightarrow \tau$
$\eta \in$ Bool type $\rightarrow \beta \mid \mathsf{F} \mid \mathsf{T} \mid \beta_1 \vee \ldots \vee \beta_n$
$P \in$ Constraint set $\rightarrow \{p_1, \ldots, p_n\}$
$p \in$ Constraint $\rightarrow \eta_1 \leq \eta_2$
$\alpha \in$ Raw type var, $\beta \in$ Bool type var

Figure 6: Syntax of types and constraints

3. THE ANALYSIS

In this section we present an example modular cloning analysis in the form of an annotated type system. This formulation is somewhat abstract in that it does not specify a translation from λ^{in} to λ^{mc}. Instead it allows us to check that a λ^{mc} program is well-typed and annotated safely. The type system can then be seen as a specification of the analysis and serves as a stepping stone in the correctness proof for the inference algorithm given in the next section. That algorithm does give a translation and is correct with respect to the type system since it either rejects the λ^{in} program or translates it to a well-typed λ^{mc} program.

3.1 Type based program analysis

We will here give a very brief introduction to type systems and type based analysis for the benefit of readers not acquainted with this subject.

One way to understand type based analysis (and type systems in general) is to start from a language with an untyped semantics which, like the semantics of λ^{mc}, has a universal set of values (closures plus error in λ^{mc}). Types then correspond to subsets of this universe. The correspondence generalizes to typing environments, which are mappings from variables to types, so that $\rho : A$ if $\rho(x) : A(x)$ for all $x \in dom(A)$. If the type system is *semantically sound*, it is then possible to prove that, if a typing judgement $A \vdash e : \tau$ (which is read as "e has type τ in typing environment A") is derivable, $\rho : A$ and $\rho \vdash e \Downarrow w$, then $w : \tau$. If the semantics models dynamic type errors using a special error value (e.g. $\rho \vdash 3\ x \Downarrow \mathsf{error}$) which is not part of any type, then the type system is a kind of program analysis that can determine that some progams do not evaluate to error (this is sometimes called a *safety analysis*).

In general, a type system defined using inference rules can assign the same expression many different types. The reason for this is, in most systems, that the inference relation is closed under substitution (a substitution θ is a function from types to types which replaces type variables in the argument type with types). If $A \vdash e : \tau$, then $\theta A \vdash e : \theta\tau$ for any substitution θ. A classic example is the identity function $id = \lambda x.x$ for which all types of the form $\tau \rightarrow \tau$ are derivable.

This might look inconvenient from the point of view of program analysis: How many times do we need to analyse id and how do we represent an infinite number of types? Here polymorphism comes to the rescue by allowing us to infer the type $\forall\alpha.\alpha \rightarrow \alpha$, where α is a *type variable*, for id. This *type scheme* succintly captures all the types of id: A type τ can be inferred for id precisely if it can be formed by substituting some τ' for α in $\alpha \rightarrow \alpha$. Since any such type is derivable, and if the type system is semantically sound, the identity function will be an element of all of these types. Thus the interpretation of a type scheme is the intersection of all of the types that can be formed from it.

We infer polymorphic types by first inferring a monomorphic type with type variables ($A \vdash \lambda x.x : \alpha \rightarrow \alpha$) and then *generalizing* over some type variables which do not occur in A (in this case α). This restriction is important for semantic soundness; suppose we have $\rho : A$, $A \vdash e : \forall\alpha.\tau$ and $\rho \vdash e \Downarrow w$. We now have to prove that $w : [\tau'/\alpha]\tau$ for all τ'. We know that in order to derive the type $\forall\alpha.\tau$, we must have derived the type τ so we have $A \vdash e : \tau$. We can now use the closure under substitution to derive $[\tau'/\alpha]A \vdash e : [\tau'/\alpha]\tau$. The above mentioned restriction now comes into play: since α does not occur in A, we have $[\tau'/\alpha]A = A$, so, since we have $\rho : A$, $\rho \vdash e \Downarrow w$ and $A \vdash e : [\tau'/\alpha]\tau$ for arbitrary τ', we have showed that $w : [\tau'/\alpha]\tau$ for arbitrary τ, as required.

3.2 Syntax of types

In the type system of a programming language, or for safety analysis, we are interested in whether values are functions, numbers, data structures and so on. When doing program analysis, we typically want to know more; in the present case, we are also interested in whether the objects are evaluated (whnf closures) or unevaluated (thunks) since this is necessary to know in order to determine if an `eval` is cheap and safe or not. In other analyses, we might be interested in whether the value is shared or how it is represented.

We will keep this information in annotations; thus values are described by *annotated types* τ of the form (η, ν) where ν is the ordinary type and η is an annotation indicating whether the value might be a thunk. A *bool type* η is either a variable β, a type constant F or T or a disjunction of variables $\beta_1 \vee \ldots \vee \beta_k$.

Ordinary types ν are the familiar type variables α, base types U (*e.g.* `Int`), function types $\tau_1 \rightarrow \tau_2$ (note that τ_1 and τ_2 are annotated types to indicate the evaluatedness of the argument and result) but also clone types of the form $\vec{\eta} \Rightarrow \tau$. These are the types of cloning abstractions, in analogy with function types being the types of lambda abstractions. Figure 6 gives the syntax of types

There is a close correspondence between annotations and annotation types in that the boolean values are also types. Thus T and F are both boolean annotations and boolean types. Annotation types can therefore be used for a very precise dataflow analysis of annotation values.

So for instance, $(\mathsf{F}, \texttt{Int})$ is the type of evaluated integers and $(\mathsf{T}, (\mathsf{T}, \texttt{Int}) \rightarrow (\mathsf{F}, \texttt{Int}))$ is the type of possibly unevaluated functions taking possibly inevaluated integers to definitely evaluated integers.

We will use *substitutions*, which are functions mapping types to types, ranged over by θ. Substitutions are entirely determined by what they map variables to. We write the substitution mapping α to ν and all other variables to themselves as $[\nu/\alpha]$ (and similarly for annotation types). Since disjunctions of the form $\beta_1 \vee \ldots \vee \beta_k$ only admit variables,

$$P \cup \{\beta \le \eta\} \Vdash \beta \le \eta \qquad \text{elem}$$

$$P \Vdash \beta \le \beta \qquad \beta\text{-reflex}$$

$$P \Vdash \mathsf{F} \le \eta \qquad P \Vdash \eta \le \mathsf{T} \qquad \text{false,true}$$

$$\frac{P \Vdash \beta_1 \le \eta \quad \ldots \quad P \Vdash \beta_n \le \eta}{P \Vdash \beta_1 \vee \ldots \vee \beta_n \le \eta} \qquad \vee\text{-left}$$

$$\frac{\beta' \in \{\beta_1, \ldots, \beta_n\} \qquad P \Vdash \beta \le \beta'}{P \Vdash \beta \le \beta_1 \vee \ldots \vee \beta_n} \qquad \vee\text{-right}$$

Figure 7: Constraint entailment

substitutions simply disjunctions when. In paricular, variables mapped to F are dropped and if some variable in the disjunction is mapped to T, the whole disjunction is mapped to T. The $\sqcup$ operator simplifies the result in the same way.

The type system uses *type constraints* p. These are inequality constraints of the form $\eta_1 \le \eta_2$ dealing with the boolean order $\mathsf{F} \le \eta \le \mathsf{T}$. Note that this is *not* a subtype ordering: The type F corresponds to the set $\{\mathsf{F}\}$ and T corresponds to $\{\mathsf{T}\}$ and $\{\mathsf{F}\} \not\subseteq \{\mathsf{T}\}$.

Constraints are related by an *entailment* relation, given in figure 7. The intention is that if a constraint set P entails a constraint p, $P \Vdash p$, then whenever the constraints in P are satisfied, the constraint p will also be satisfied. Constraints that are entailed by the empty set, $\emptyset \Vdash p$, are called *tautological*. Examples include $\mathsf{F} \le \eta$, $\beta \le \beta \vee \beta'$ and many more.

Polymorphism is expressed using type schemes of the form $\forall\vec{\beta}, \vec{\alpha}.\tau \,|\, P$ where the type variables in $\vec{\beta}$ and $\vec{\alpha}$ are universally quantified and P constrains the possible instantiations of the $\vec{\beta}$ and $\vec{\alpha}$. So a value is an element of the polymorphic type $\forall\vec{\beta}, \vec{\alpha}.\tau \,|\, P$ if it is an element of every $\theta(\tau)$ such that $\emptyset \Vdash \theta P$ and $\theta = [\vec{\eta}/\vec{\beta}, \vec{\nu}/\vec{\alpha}]$ for some $\vec{\eta}$ and $\vec{\nu}$. Type schemes always represent sets of types; constraints provide more fine-grained control than conventional type schemes. Constrained type schemes are used in many other systems, for instance in the theory of qualified types [15]. We will write the type scheme $\forall.\tau \,|\, \emptyset$ simply as τ.

3.3 Type schemes and context dependence

In our type based analysis, type schemes are instrumental in capturing the dependence on the unknown context. They play the role that *summary functions* play in some interprocedural data flow analysers for imperative languages [4].

When discussing examples, the syntax given for types in figure 6 is rather awkward so we will use a prettier alternative syntax: We will write $(\eta, \tau_1 \to \tau_2)$ as $\tau_1 \to^\eta \tau_2$ (and similarly for $\Rightarrow$) and move the annotation to a superscript on ordinary type variables α and basic types U.

To see how a type scheme captures context dependence, consider the identity function, defined by `id=λx. eval x` in λ^{in}(the `eval` ensures that the return value of the function is not a thunk). Note that the `eval` is redundant if the argument to `id` is statically known to be evaluated. Since this is a context dependent property, the analyser transforms this binding into `id=`$\{\mathtt{u}\}$`λx.eval`$\{\mathtt{u}\}$` x` and derives the type scheme $(\forall a, u, v, t, s.\langle u\rangle \Rightarrow^v a^u \to^t a^s \,|\, \emptyset)$ for the (transformed) binding.[3] The $\langle u\rangle \Rightarrow^v$ part signifies that the (transformed) `id` is a cloning function representing a set of different versions of the identity function, one of which must be selected by applying it to a boolean annotation value of type u. The boolean type variable u can be instantiated to T or F; the inference algorithm will choose F if possible, but since u also occurs as an annotation on the second argument of `id` this is only possible if that argument can be given a type of the form ν^{F}, *i.e.* the type of an evaluated value.

The analyser will transform each occurrence of `id` to a well typed cloning application; thus if u is instantiated to T, that occurrence of `id` will be applied to T, the only boolean expression of type T.

As a further illustration, the types of the λ^{mc} version of the example functions (from figure 2) are shown in figure 8. The type for `libfun` tells us that it is a polymorphic cloning function with two cloning parameters (with annotation types t and u) that returns a function taking two arguments, a function and an integer, and returning a value of whatever type the functional argument returns (a^w). Further, the first cloning parameter has the same type (t) as the evaluatedness annotation on the functional argument (`f`). This captures the fact that the `eval` of `f` in `libfun` (annotated with $\mathtt{u}_1$) can be eliminated only if the second argument (`n`) is already evaluated. The second cloning parameter of `libfun` has the same type (u) as the evaluatedness annotation on the integer argument `n`. Finally, the second cloning parameter must be smaller than the evaluatedness annotation on the argument part of the function ($u \le v$), prohibiting instantiation of u to T and v to F. This is an example of how the type-based approach deals with indirect function calls.

3.4 Inference rules

The inference rules for typing λ^{mc} modules and expressions are given in Figure 9. The judgements for expressions are of the form $P, A, \eta \vdash e : \tau$ where P is a set of constraints, A is a typing environment associating variabels x with type schemes σ and boolean variables z with boolean types η, η is a boolean type giving the evaluation budget of the expression (F if the expression must be cheap and T if it is allowed to be expensive), e is a λ^{mc} expression and τ is the type of the values that e might evaluate to. We extend typing environments from annotation variables to annotation expressions in the same way as for value environments ($A(\mathsf{F}) = \mathsf{F}$ and $A(\mathsf{T}) = \mathsf{T}$).

For module sequences, we have judgements of the form $P, A \vdash ms$ `ok` which states that ms is well-typed in the typing environment A if the constraints P are satisfied.

For operator applications the inference system uses *operator axioms* of the form $\vdash op\ \tau_1 \ldots \tau_k : \nu, \eta$ which are closed

[3] The boolean type variables v, t and s are only technically necessary since the system in this paper does not have subtypes; think of them as F in this context.

$$\texttt{libfun} : \forall a,t,u,v,w,q,r,s.\langle t,u\rangle \Rightarrow^q (\texttt{Int}^v \rightarrow^t a^w) \rightarrow^r \texttt{Int}^u \rightarrow^s a^w \,|\, \{u \leq v\}$$
$$\texttt{appfun} : \forall t,u.\langle t,u\rangle \Rightarrow^v (\texttt{Int}^u \rightarrow^t \texttt{Int}^w) \rightarrow^q \texttt{Int}^u \rightarrow^r \texttt{Int}^s \,|\, \emptyset$$

Figure 8: The types of `libfun` and `appfun`

under substitution (if we have $\vdash\ op\ \tau_1 \ldots \tau_k : \nu, \eta$, we also have $\vdash\ op\ \theta\tau_1 \ldots \theta\tau_k : \theta\nu, \theta\eta$ for any substitution θ).

The rules are syntax directed; generalization is built into the [module] and [let] rules and instantiation is built into the [var] rule. Since an expression that always produces an evaluated value, such as a lambda expression, should be usable in a context expecting a thunk, the conclusions of the rules for such expressions (*e.g.* the [abs] rule) has an arbitrary boolean type as evaluatedness annotation (η' does not occur elsewhere in the [abs] rule).

Most expressions can be evaluated using the restricted evaluation budget F if their subexpressions are also cheap. The exception is applications; in the [app] rule, the conclusion has evaluation budget T.

The core of the analysis is found in the [thunk] and [eval] rules. In the [thunk] rule, the body must be typable with an evaluation budget corresponding to the annotation a on the `thunk`. Thus for the thunk to be eliminated ($A(a) = \mathsf{F}$), its body must be cheap. The constraint premise expresses the condition that if the annotation is T, then the result of the expression is unevaluated.

The [eval] rule says that if the argument is unevaluated, the annotation and the evaluation budget of the expression must both be T.

4. AN INFERENCE ALGORITHM

In this section we turn the inference rules of Section 3 into an inference algorithm. This algorithm translates a λ^{in} program to a λ^{mc} program by adding annotations as well as cloning abstractions and applications. In order to make it possible to compile the cloning using code duplication, only right-hand-sides of bindings are ever cloned. Every occurrence of a variable bound to a cloned expression is translated to a cloning application. In this way, cloning abstractions occur only where the inference algorithm generalizes and cloning applications occur where type schemes are instantiated (although not all binings are cloned). The generated program is guaranteed to be typable in the λ^{mc} type system.

The main part of the algorithm is the function Inf, defined together with some auxilliary functions in figure 10, which takes a typing environment A and an expression e and returns a substitution θ, a constraint set S, a boolean type η, an annotated type τ and a λ^{mc} expression e'.

The functin $\mathsf{InfMods}$ processes module sequences from left to right analysing each module, accumulating the returned constraints and recording the types derived for the exported variables. When all modules have been processed, the accumulated constraints are simplified.

$\boxed{P, A \vdash ms\ \texttt{ok}}$

$$\frac{P', A, \mathsf{T} \vdash e : \tau \qquad (\vec{\beta} \cup \vec{\alpha}) \cap fv(A) = \emptyset \qquad P, A[x \mapsto \forall\vec{\beta},\vec{\alpha}.\tau \,|\, P'] \vdash ms\ \texttt{ok}}{P, A \vdash \texttt{module}\ x = e;\ ms\ \texttt{ok}} \quad \text{module}$$

$$P, A \vdash \epsilon\ \texttt{ok} \quad \text{main}$$

$\boxed{P, A, \eta \vdash e : \tau}$

$$\frac{A(x) = \forall\vec{\beta},\vec{\alpha}.\tau \,|\, P' \qquad \theta = [\vec{\eta}/\vec{\beta},\vec{\nu}/\vec{\alpha}] \qquad P \Vdash \theta P'}{P, A, \eta \vdash x : \theta\tau} \quad \text{var}$$

$$\frac{P, A, \mathsf{T} \vdash e_1 : (\mathsf{F}, \tau' \rightarrow \tau) \qquad P, A, \mathsf{T} \vdash e_2 : \tau'}{P, A, \mathsf{T} \vdash e_1\, e_2 : \tau} \quad \text{app}$$

$$\frac{P, A[x \mapsto \tau'], \mathsf{T} \vdash e : \tau}{P, A, \eta \vdash \lambda x.\, e : (\eta', \tau' \rightarrow \tau)} \quad \text{abs}$$

$$\frac{\vdash op\,\tau_1 \ldots \tau_r : \nu, \eta \qquad P, A, \eta \vdash e_1 : \tau_1 \quad \ldots \quad P, A, \eta \vdash e_r : \tau_r}{P, A, \eta \vdash op\, e_1 \ldots e_r : (\eta', \nu)} \quad \text{op}$$

$$\frac{P', A, \eta \vdash e' : \tau' \qquad (\vec{\beta} \cup \vec{\alpha}) \cap fv(A, \eta) = \emptyset \qquad P, A[x \mapsto \forall\vec{\beta},\vec{\alpha}.\tau' \,|\, P'], \eta \vdash e : \tau}{P, A, \eta \vdash \texttt{let}\ x = e'\ \texttt{in}\ e : \tau} \quad \text{let}$$

$$\frac{P, A, A(a) \vdash e : (\mathsf{F}, \nu) \qquad P \Vdash A(a) \leq \eta'}{P, A, \eta \vdash \texttt{thunk}\{a\}\, e : (\eta', \nu)} \quad \text{thunk}$$

$$\frac{P, A, \eta \vdash e : (\eta'', \nu) \qquad P \Vdash \{\eta'' \leq A(a),\ \eta'' \leq \eta\}}{P, A, \eta \vdash \texttt{eval}\{a\}\, e : (\eta', \nu)} \quad \text{eval}$$

$$\frac{P, A[\vec{z} \mapsto \vec{\eta}], \mathsf{F} \vdash b : \tau}{P, A, \eta \vdash \{\vec{z}\}\ b : (\eta', \vec{\eta} \Rightarrow \tau)} \quad \text{clone}$$

$$\frac{P, A, \eta \vdash e : (\mathsf{F}, \vec{\eta} \Rightarrow \tau) \qquad A(\vec{a}) = \vec{\eta}}{P, A, \eta \vdash e\{\vec{a}\} : \tau} \quad \text{inst}$$

Figure 9: The inference system

The algorithm differs from the inference system in that information about annotation variables is part of the constraints returned (S) rather than the typing environment A. The information takes the form of associations $z : \eta$. This difference is a consequence of the absence of cloning abstractions in the original program; the analyser invents annotation variables for the translated expression as it analyses the input expression. We write $\mathsf{Assoc}(S)$ for the associations in S and $\mathsf{Con}(S)$ for the constraints.

The inference algorithm is related to the type system by syntactic soundness, meaning that the result of the algorithm is always derivable in the inference system.

4.1 Analysing expressions

Inf is mainly the usual adaptation of algorithm W [8] to systems with constrained (or qualified) types (see *e.g.* Jones [15] for a similar treatment). In particular, it returns a substitution θ which carries additional information about the types of the free variables of e found during type checking of e. This solves the problem that the type of a variable bound in a lambda abstraction $\lambda x.e$ is not known when that type must be recorded in the typing environment A for the recursive call to infer a type for e. We assume a fresh annotated type (of the form (β, α) where β and α are fresh variables) for the bound variable x. When Inf returns after checking the body e, the returned substitution θ gives the type of x as $\theta(\beta, \alpha)$.

The substitutions are constructed by a standard unification algorithm *mgu* which we also use for unifying sequences of type expressions.

When analysing a variable occurrence x, the type information is consulted to determine if the variable is bound to a cloning abstraction, in which case the occurrence should be translated to a cloning application $x\{\vec{z}\}$. In that case, a fresh set of annotation variables $\vec{z}$ are used, just as is done for other annotated constructs (`thunk` and `eval` expressions).

The cases for application and abstraction are rather standard, and follow from the corresponding inference rules. The case for operators uses operator axioms, freshly renamed. The case for `let` expressions relegates the gory details of generalization and cloning to the InfRHS function. The cases for `thunk`s and `eval`s annotate the constructs with fresh annotation variables. Note that a `thunk` expression is always cheap and safe; either the body is cheap and safe or we will definitely not speculate the thunk.

4.2 Analysing right hand sides

The function InfRHS infers types for right hand sides in bindings. In four lines it infers a type for the expression, simplifies the constraints, decides about cloning and generalizes.

Simplifying the constraints is important since type schemes are in general instantiated multiple times. Also, it makes the files with analysis information more compact. The simplifications performed by Simplify and Clone in effect find the optimizations that are legal regardless of the properties of the calls of the function.

The simplifications implemented by Simplify consist in com-

$\mathsf{InfMods}(S, imods, A, (\texttt{module}\ x = e;\ smods))$
$= \mathsf{InfMods}(\theta S \cup S', (imods; \texttt{module}\ x = e'), \theta A[x \mapsto \sigma], smods)$
where $(\theta, S', \eta, \sigma, e') = \mathsf{InfRHS}(A, e)$
$\mathsf{InfMods}(S, imods, A, \epsilon) = \mathsf{if}\ S_c = \emptyset\ \mathsf{then}\ ([\vec{z} \mapsto \underline{\vec{a}}], imods)\ \mathsf{else\ fail}$
where $S_c \cup \{\vec{z} : \underline{\vec{a}}\} = \mathsf{Simplify}(\emptyset, S)$

$\mathsf{Inf}(A, x) = \mathsf{case}\ \tau\ \mathsf{of}$
$(\eta, \vec{\eta} \Rightarrow \tau') \rightarrow (id, \theta(S \cup \{\vec{z} : \vec{\eta}\}), \mathsf{F}, \theta(\tau'), x\{\vec{z}\})$
$\tau' \rightarrow (id, \theta(S), \mathsf{F}, \theta(\tau'), x)$
where $(\forall \vec{\beta}, \vec{\alpha}.\tau \mid S) = A(x)$
$\theta = [\vec{\beta}'/\vec{\beta}, \vec{\alpha}'/\vec{\alpha}]$
$\vec{\beta}', \vec{\alpha}'$ fresh
$\mathsf{Inf}(A, e_1\ e_2) = (\theta \circ \theta_2 \circ \theta_1, \theta(\theta_2 S_1 \cup S_2), \mathsf{T}, \theta\tau, e_1'\ e_2')$
where $(\theta_1, S_1, \eta_1, \tau_1, e_1') = \mathsf{Inf}(A, e_1)$
$(\theta_2, S_2, \eta_2, \tau_2, e_2') = \mathsf{Inf}(\theta_1 A, e_2)$
$\theta = mgu(\tau_1, (\mathsf{F}, \tau_2 \rightarrow \tau))$
τ fresh
$\mathsf{Inf}(A, \lambda x.\, e) = (\theta, S, \mathsf{F}, (\beta, \theta\tau \rightarrow \tau'), \lambda x.\, e')$
where $(\theta, S, \eta, \tau', e') = \mathsf{Inf}(A[x \mapsto \tau], e)$
τ, β fresh
$\mathsf{Inf}(A, op\ e_1 \ldots e_k) = (\theta', S, \eta', (\beta, \nu), op\ e_1' \ldots e_k')$
where $(\theta_1, S_1, \eta_1, \tau_1', e_1') = \mathsf{Inf}(A, e_1)$
$\ldots$
$(\theta_k, S_k, \eta_k, \tau_k', e_k') = \mathsf{Inf}((\theta_{k-1} \circ \ldots \circ \theta_1)A, e_2)$
$\vdash op\ \tau_1 \ldots \tau_k : \nu, \eta$ fresh
$\theta = mgu([\tau_1, \ldots, \tau_k], [(\theta_k \circ \ldots \circ \theta_2)\tau_1', \ldots, \tau_k'])$
$S = (\theta \circ \theta_k \circ \ldots \circ \theta_2)S_1 \cup \ldots \cup \theta S_k$
$\eta' = (\theta \circ \theta_k \circ \ldots \circ \theta_2)\eta_1 \sqcup \ldots \sqcup \theta\eta_k \sqcup \theta\eta$
$\theta' = \theta \circ \theta_k \circ \ldots \circ \theta_1$
β fresh
$\mathsf{Inf}(A, \texttt{let}\ x = e_1\ \texttt{in}\ e_2) = (\theta_2 \circ \theta_1, S, \theta_2\eta_1 \sqcup \eta_2, \tau, \texttt{let}\ x = e_1'\ \texttt{in}\ e_2')$
where $(\theta_1, S_1, \eta_1, \sigma, e_1') = \mathsf{InfRHS}(A, e_1)$
$(\theta_2, S_2, \eta_2, \tau, e_2') = \mathsf{Inf}((\theta_1 A)[x \mapsto \sigma], e_2)$
$S = \theta_2 S_1 \cup S_2$
$\mathsf{Inf}(A, \texttt{thunk}\ e) = (\theta, S \cup \{z : \beta, \eta \leq \beta, \eta' \leq \mathsf{F}\}, \mathsf{F}, (\beta, \nu), \texttt{thunk}\{z\}\ e')$
where $(\theta, S, \eta, (\eta', \nu), e') = \mathsf{Inf}(A, e)$
β, z fresh
$\mathsf{Inf}(A, \texttt{eval}\ e) = (\theta, S \cup \{z : \beta, \eta' \leq \beta\}, \eta \sqcup \eta', (\beta', \nu), \texttt{eval}\{z\}\ e')$
where $(\theta, S, \eta, (\eta', \nu), e') = \mathsf{Inf}(A, e)$
z, β, β' fresh

$\mathsf{InfRHS}(A, e) = (\theta, S_a, \eta', (\forall \vec{\beta}, \vec{\alpha}.\tau' \mid \mathsf{Con}(S')), e'')$
where $(\theta, S, \eta, \tau, e') = \mathsf{Inf}(A, e)$
$S' = \mathsf{Simplify}(fv(\theta A, \eta, \tau), S)$
$(S_a, \eta', \tau', e'') = \mathsf{Clone}(\mathsf{Assoc}(S'), \eta, \tau, e')$
$\vec{\beta}, \vec{\alpha} = fv(\mathsf{Con}(S'), \tau') \setminus fv(\theta A, \eta', S_a)$

$\mathsf{Simplify}(W, S) = \theta S$
where $\theta(\beta) = \begin{cases} \sqcup\{\eta \mid \eta \in W \cup \{\mathsf{T}\} \wedge \eta \leq_S^+ \beta\}, & \text{if } \beta \notin W \\ \beta, & \text{otherwise} \end{cases}$

$\mathsf{Clone}(S_a, \eta, \tau, e)$
$= \mathsf{if\ clone\ then}\ (\emptyset, (\beta', \mathsf{F}, \langle \eta_1, \ldots, \eta_n \rangle \Rightarrow \tau), \{z_1, \ldots, z_n\}\,\Theta(e))$
$\mathsf{else}\ (\{z_1 : \eta_1, \ldots, z_n : \eta_n\}, \eta', \tau, \Theta(e))$
where $\{\eta_1, \ldots, \eta_n\} = \{\eta \mid \exists z.z : \eta \in S_a\} \setminus \{\mathsf{F}, \mathsf{T}\}$
$\Theta(z) = \begin{cases} \underline{a}, & \text{if } z : \underline{a} \in S_a \\ z_i, & \text{if } z : \eta_i \in S_a \end{cases}$
$z_1, \ldots, z_n$ fresh

Auxilliary definitions:

$\eta_1 \leq_S^+ \eta_2$ iff $\eta_1 \leq \eta_2 \in S$ or $\exists \eta.\eta_1 \leq \eta \in S \wedge \eta \leq_S^+ \eta_2$

$\mathsf{Assoc}(S) = \{z : \eta \mid z : \eta \in S\}$
$\mathsf{Con}(S) = \{\eta \leq \eta' \mid \eta \leq \eta' \in S\}$

$\mathsf{NonTriv}(S) = \mathsf{Assoc}(S) \cup \{p \mid p \in \mathsf{Con}(S) \wedge \emptyset \not\Vdash p\}$

Figure 10: The inference algorithm

puting a substitution, applying it to the constraints and removing tautological constraints from the result. It is here that the boolean type variable disjunctions are used; if the constraints are $\{\beta_1 \leq \beta, \beta_2 \leq \beta\}$ with $W = \{\beta_1, \beta_2\}$, the substitution will map β to $\beta_1 \vee \beta_2$.

The function Clone performs cloning and cloning-related simplifications. It is called with a set S_a of associations, a cost η and type τ and an expression e (already translated) where η is the cost and τ is the type of e. It returns a new set of associations S'_a, a new cost τ' and type τ' and a new expression e'. The input associations S_a have been passed through simplification, so some of the variables may now be associated with annotation values ($z : \underline{a}$), meaning that the corresponding optimization conditions are now known. These variables are unnecessary and they can be replaced in e' by the associated values. This is the mechanism by which the analyser finds those transformations which do not depend on the context.

In addition, different annotation variables may be associated with the same annotation type ($z_1 : \eta, z_2 : \eta$), if the associated optimization conditions are still unknown but identical. In that case, these annotation variables (z_1 and z_2) will always have the same values and can be replaced by a single variable. This simplification can be said to find the "degrees of freedom" of the cloned binding, those dimensions along which the different version may differ. The substitution Θ, which maps annotation variables to annotations, contains the information obtained from these two simplifications.

It is in this step that the analyser finds that the `thunk` and the `eval` of `n` in the function `libfun` (see figures 1 and 2) should be annotated with the same annotation variable and that the F, F-version of `libfun` should be used in the first application in `appfun`.

It is crucially important for the performance of the analyser to apply these two simplifications since *all* annotation variables which the algorithm invents for an expression would otherwise turn up in the cloning abstractions, leading to exponential code growth not only in the worst but in the common case, since the number of annotations in a cloning abstraction would be proportional to the number of `thunk`s and `eval`s in the cloned function plus the number of variables occurring in cloning applications.

Next, Clone decides whether this expression is suitable for cloning. As a necessary condition, the expression must be a normal form, that is an abstraction or a constant. Typically, we are interested in cloning functions, but if functions are embedded in data structures (in richer languages than λ^{in}), it might be a good idea to clone data structures too, in order to be able to clone the functions inside. Cloning may also depend on whether the expression is part of a top level definition or if it is nested. In the latter case it might be more expensive to clone since a nested function may have free non global variables so that each clone is represented by its own dynamically allocated closure.

We leave the condition unspecified in the definition of Clone but note that a reasonable strategy for λ^{in} is to clone only top level functions (bindings with lambda abstractions as right hand sides).

$$
\begin{array}{l}
\mathsf{TrMods}(\rho, tmods, V, (imods; \texttt{module}\ x = e)) \\
\quad = \mathsf{TrMods}(\rho, (\texttt{module}\ bnd; tmods), V \cup fv(bnd), imods) \\
\quad \text{where } bnd = \mathsf{TrBind}(\rho, V, x = e) \\
\mathsf{TrMods}(\rho, tmods, V, \epsilon) = tmods \\
\\
\mathsf{Tr}(\rho, x) = x \\
\mathsf{Tr}(\rho, x\{\vec{a}\}) = x\_\rho(\vec{a}) \\
\mathsf{Tr}(\rho, e_1\ e_2) = \mathsf{Tr}(\rho, e_1)\ \mathsf{Tr}(\rho, e_2) \\
\mathsf{Tr}(\rho, \lambda x.\, e) = \lambda x.\, \mathsf{Tr}(\rho, e) \\
\mathsf{Tr}(\rho, op\, e_1 \ldots e_k) = op\ \mathsf{Tr}(\rho, e_1) \ldots \mathsf{Tr}(\rho, e_k) \\
\mathsf{Tr}(\rho, \texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \texttt{let}\ \mathsf{TrBind}(\rho, fv(e'_2), x = e_1)\texttt{in}\ e'_2) \\
\quad \text{where } e'_2 = \mathsf{Tr}(\rho, e_2) \\
\mathsf{Tr}(\rho, \texttt{thunk}\{a\}\, e) = \mathsf{if}\ \rho(a) = \mathsf{T}\ \mathsf{then}\ \texttt{thunk}\ \mathsf{Tr}(\rho, e)\ \mathsf{else}\ \mathsf{Tr}(\rho, e) \\
\mathsf{Tr}(\rho, \texttt{eval}\{a\}\, e) = \mathsf{if}\ \rho(a) = \mathsf{T}\ \mathsf{then}\ \texttt{eval}\ \mathsf{Tr}(\rho, e)\ \mathsf{else}\ \mathsf{Tr}(\rho, e) \\
\\
\mathsf{TrBind}(\rho, V, x = \{\vec{z}\}\, b) = \begin{pmatrix} (x\_\underline{\vec{a}}_1) = \mathsf{Tr}(\rho[\vec{z} \mapsto \underline{\vec{a}}_1], b) \\ \ldots \\ (x\_\underline{\vec{a}}_n) = \mathsf{Tr}(\rho[\vec{z} \mapsto \underline{\vec{a}}_n], b) \end{pmatrix} \\
\quad \text{where } \{\underline{\vec{a}}_1, \ldots, \underline{\vec{a}}_n\} = \{\underline{\vec{a}} \mid x\_\underline{\vec{a}} \in V\} \\
\mathsf{TrBind}(\rho, V, x = e) = x = \mathsf{Tr}(\rho, e)
\end{array}
$$

Figure 11: The specializer

5. SPECIALIZATION

When the program has been translated to λ^{mc}, we can use specialization to remove all of the cloning constructs and make all annotations ground (variable free). The result of this translation is a program in λ^{out} that can be translated to assembly (or some other target language) by a conventional code generator.

The analysis and translation algorithm discussed in section 4 generates cloning abstractions only as right hand sides of `let` expressions (`let` $x = \{\vec{z}\}\, b$ `in` e) where every occurrence of x in e is in a cloning application $x\{\vec{a}\}$. The specializer first traverses e, translating it to e', replacing every cloning application $x\{\vec{a}\}$ with a reference to a new variable formed from the cloning variable x and the value $\underline{\vec{a}}$ of the argument (at this point, the values of all of the annotation variables are known). We write $x\_\underline{\vec{a}}$ for the new variable. This mechanism is called *mangling* and is used in many contexts, for instance in the implementation of C++ where overloaded identifiers have (target language) names which encode type information. We use it to ensure that a unique variable name is generated for each distinct argument value that x is applied to.

Having translated the body e, the specializer also collects all arguments $\{\underline{\vec{a}}_1, \ldots, \underline{\vec{a}}_n\}$ that x is applied to, that is, all $\underline{\vec{a}}$ such that the translation of e has $x\_\underline{\vec{a}}$ as a free variable. For each $\underline{\vec{a}}_i$ a copy b_i of the body b of the cloning abstraction is made with the annotation variables $\vec{z}$ bound to $\underline{\vec{a}}_i$. Finally, the original binding is replaced by n new bindings of the form $(x\_\underline{\vec{a}}_i) = b_i$. The tranlsation is performed by the function Tr, defined in figure 11, which takes an environment ρ, mapping annotation variables to annotation values, and an expression e and returns a translated expression e'. The auxilliary function TrBind takes an environment ρ, a set V of free variables of the body and a binding $x = e$ and returns a new binding (which might bind several variables if the right hand side of the binding is a cloning abstraction).

The function TrMods specializes module sequences from right to left, starting with the `main` module. TrMods has an accumulating parameter V which collects the free variables of the target (λ^{out}) modules generated so far. This set is used to determine which specialized versions to make of the top-level binding in each module.

The syntax of the target language of specialization differs from that of λ^{mc} by allowing for mangled variables and `let` expressions binding several variables as well as by the absence of cloning abstractions and applications as well as annotation variables (all annotations are annotation values).

Note that this specializer does not do full monomorphization. For instance, at most two versions of the identity function will be produced, with and without the `eval`. Both of these versions will be polymorphic in the ordinary type of the argument. For representation selection, where some types may get specialized representations, the boxed versions will still be polymorphic. A consequence of this fact is that all versions of a cloned binding will get different code.

6. SELECTIVE RECOMPILATION

Systems supporting separate compilation typically also do not need to recompile all modules when one source file has been edited. This is a very useful feature during program development, even if one might imagine compiling the program without context dependent optimizations most of the time, only wielding the really big hammer occasionally. Our system however does support recompiling a subset of the modules after an editing change even when applying full context dependent optimization.

Specifically, our algorithm for selective recompilation implements the following strategy for the first pass:

- If a λ^{in} module has changed, it must be reanalysed.
- If reanalysis leads to attribute information (the substitution, constraint set and type scheme returned by InfRHS) that differs from that given by the previous run of the analyser, then all λ^{in} modules importing the reanalysed module must be reanalysed.

For the second pass we have:

- If a λ^{in} module has been reanalysed in the first pass, the regenerated λ^{mc} module must be respecialized.
- If a regenerated λ^{out} module needs a clone of an imported entity that is not provided by the exporting λ^{out} module, the exporting λ^{mc} module must be respecialized. In that case, the union of the previously provided clones and the newly requested clones are generated.

Note that the respecialization (second pass) strategy keeps generating all clones it has ever generated for a particular module. This is deliberate, with the intention that, during development, the program will reach a steady state with respect to which clones are needed, limiting the recompilations triggered by missing versions.

$$
\begin{array}{l}
\mathsf{InfMods}(IL, At, \theta, Tgt, (x; xs)) \\
\quad = \mathsf{if}\ x \in dom(IL) \\
\qquad \mathsf{then}\ \mathsf{InfMods}(IL, At, \mathsf{GetSub}(At, x) \circ \theta, Tgt, vs) \\
\qquad \mathsf{else}\ \mathsf{InfMods}(IL'', Attr', \theta' \circ \theta, Tgt', vs) \\
\quad\ \ \mathsf{where}\ (\texttt{module}\ x = e) = Src(x) \\
\qquad (\theta', S, \eta, \sigma, e') = \mathsf{InfRHS}(\theta\mathsf{GetEnv}(At, fv(e)), e) \\
\qquad At' = At[x \mapsto (\sigma, S, \theta')] \\
\qquad IL' = \mathsf{if}\ At = At'\ \mathsf{then}\ IL\ \mathsf{else}\ IL \setminus \{x' \mid x \in fv(Src(x'))\} \\
\qquad IL'' = IL'[x \mapsto \texttt{module}\ x = e'] \\
\qquad Tgt' = Tgt \setminus \{x\} \\
\mathsf{InfMods}(IL, At, \theta, Tgt, \epsilon) = \mathsf{if}\ S_c = \emptyset\ \mathsf{then}\ ([\vec{z} \mapsto \underline{\vec{a}}], IL, At, Tgt) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ fail} \\
\quad\ \ \mathsf{where}\ S_c \cup \{\vec{z} : \underline{\vec{a}}\} = \mathsf{Simplify}(\emptyset, \theta(\mathsf{GetAssoc}(At)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{GetEnv}(At, X) = [x \mapsto \sigma \mid x \in X \wedge (\sigma, S, \theta) = At(x)] \\
\mathsf{GetAssoc}(At) = \cup\{S \mid x \in dom(At) \wedge (\sigma, S, \theta) = At(x)\} \\
\mathsf{GetSub}(At, x) = \theta \\
\quad\ \ \mathsf{where}\ (\sigma, S, \theta) = At(x)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{TrMods}(\rho, IL, Vs, Tgt, (xs; x)) \\
\qquad = \mathsf{if}\ x \in dom(Tgt)\ \mathsf{then}\ \mathsf{TrMods}(\rho, IL, Vs, Tgt, xs) \\
\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ \mathsf{TrMods}(\rho, IL, Vs', Tgt', xs) \\
\quad\ \ \mathsf{where}\ (\texttt{module}\ x = e) = IL(x) \\
\qquad bnd = \mathsf{TrBind}(\rho, Vs, x = e) \\
\qquad V = fv(bnd) \\
\qquad Tgt' = Tgt[x \mapsto \texttt{module}\ bnd] \setminus \{x' \mid \exists \underline{\vec{a}}.x'\_\underline{\vec{a}} \in V \setminus Vs\} \\
\qquad Vs' = Vs \cup V \\
\mathsf{TrMods}(\rho, IL, Vs, Tgt, \epsilon) = Tgt
\end{array}
$$

Figure 12: Selective recompilation

There are two possible refinements that can be applied. First, if the attribute information changes only to be more general (better in analysis terms), then importing modules need not be recompiled. Of course, avoiding recompilation forgoes the possible benefits of the improved analysis result. Second, rather than collecting all clones generated from the same λ^{mc} module in one λ^{out} module, each clone might get its own module. In this way, respecialization is speeded up since only the newly requested versions need to be generated.

We will now turn to the details of the algorithm for selective recompilation given in figure 12. In order to keep the formalism closer to a real programming system, we will represent a modular program as a mapping from module names to modules.

The functions InfMods and TrMods use a set of accumulating parameters corresponding to the file system. These are IL (the λ^{mc} modules), At (containing the substitutions, constraints and type schemes returned by InfRHS), Vs (the free variables of the modules) and Tgt (the λ^{out} modules). The last parameter of both functions is a sequence of variables (which function as module names) that determine the processing order of the modules. The rightmost element of the sequence is always `main`. In addition, Src is treated as a global variable containing the λ^{in} modules.

A possible performance problem for InfMods is the accumulated sunstitution θ, built from compositions of the substitutions θ' returned from InfRHS. These θ' need however only record values for type variables that are free in the global typing environments (in At). The only such variables

are those that record the types of free annotation variables, that is, annotation variables occurring in modules not selected for cloning. Since the bulk of a program is made up of functions, we believe that the number of free annotation variables will be relatively limited.

This is in fact not a mere technicality; the free annotation variables reflect the fact that the optimization of the part of the program that can not be cloned can not be performed until the entire program has been processed. So in fact, cloning alleviates the problem of separate compilation in addition to improving the effectiveness of the transformations.

The first time a program is compiled, only the *Src* mapping will be defined. This will make InfMods process all modules (since *IL* not defined). TrMods will also process all modules since *Tgt* is not defined. After an editing change, the corresponding *IL* module will be removed and the program recompiled. When InfMods generates an *IL* module it removes the *Tgt* module and, if the type information has changed, the *IL* entries for importing modules. Similarly, TrMods recompiles those *IL* modules which have no corresponding *Tgt* modules, removing imported *Tgt* modules that do not export all versions that the current module uses.

7. RELATED WORK

There is a parallel in our system to the techniques [22] used for overloading in Haskell, specifically by the implementation technique of adding extra *dictionary arguments*. In fact, one could think of our `eval` and `thunk` constructs as overloaded on the annotation parts of the types. Our clone types $\vec{\eta} \Rightarrow \tau$ would then correspond to *contexts* in Haskell types. That analogy can be taken further by relating our specializations to Jones' work on removing dictionary passing by partial evaluation [16]. An interesting observation is that specialization does not in practice cause code explosion; sometimes, the programs even shrink! Other frameworks for specializing programs with respect to static properties (analysis results) are given by Consel and Khoo [5] for a functional language, and by Puebla and Hermenegildo [21] for logic languages. In both cases, abstract interpretation is assumed as the analysis framework.

Most work on partial evaluation, including Jones' above, assumes that the whole program is available at once. Dussart *et.al.* [12] presents a technique for partial evaluation of modular programs where each function is first transformed to a *generating extension* which takes the same arguments as the original function plus arguments describing *binding times*, essentially directing the generating extension as to what part of its arguments it should specialize with respect to. These binding time parameters are in a sense analogous to our annotation parameters, and it is an interesting parallel that they are also computed using a type based analysis. An important difference is that the generating extenstions are not the functions from the transformed program; they are functions that will evaluate to (intermediate representations of) these functions. The specialization itself is not done modularly; it is done by linking the modules containing the generating functions and running the linked program.

Cloning has been studied previously, both for imperative [6, 1, 17] and object oriented [3, 20, 10] languages.

The compiler literature contains some answers to the question of how to best combine (context dependent) interprocedural optimizations with (some form of) separate compilation. Several systems [17, 1, 7] use some form of *program database* containing analysis results and intermediate representation of code. The compiler repeatedly reads and writes this database during the production of an executable. All of these systems use cloning to improve the effectiveness of optimizations. The first two are production compilers (the CONVEX Application Compiler and the HP Cross Module Optimizer).

Dean *et.al* [10] are also able to avoid recompiling the entire program after an editing change by keeping track of dependencies between subprograms. However, in their case a simplifying factor is that the transformation they make (replacing method look-up with direct calls) can be performed one method (procedure) at a time since it does not depend on other procedures to be updated as well. In contrast, our system can handle the case where a set of `thunk`s and `eval`s, spread over the program, need to be eliminated or not as a unit (the same situation arises in representation selection, for instance).

The Church project, where type based flow analysis [2] is combined with a type based transformation system incorporating cloning [11], is close in spirit to this system, with the difference that they do whole-program compilation rather than our modular approach.

8. CONCLUSIONS

We have presented an integrated program analysis and transformation system which combines context sensitive program analysis with cloning and modular compilation. We have not yet implemented this system, so we have no experimental results. However, the system can easily be extended to implement the same logic as we have previously implemented in our whole program compiler. As for the particular analysis presented here, a very similar analysis has shown reductions in execution time ranging from about 5–50% for a set of small lazy functional programs.

9. REFERENCES

[1] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. In *SIGPLAN 98*, Montreal, 1998.

[2] Anindya Banerjee. A modular, polyvariant and type-based closure analysis. *ACM SIGPLAN Notices*, 32(8):1–??, August 1997.

[3] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, July 1989.

[4] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–146. ACM Press, 1999.

[5] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[6] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, February 1993.

[7] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimizations in the R(n) programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–523, October 1986.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[9] Manuvir Das, Ben Liblit, Manuel Fändrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proc. of the 8th International Symposium on Static Analysis*, 2001. LNCS 2126.

[10] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102. ACM Press, 1995.

[11] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. *ACM SIGPLAN Notices*, 32(8), August 1997.

[12] Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. *ACM SIGPLAN Notices*, 32(5):206–214, May 1997.

[13] Karl-Filip Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In Philip Wadler, editor, *Proceedings of the 2000 International Conference on Functional Programming*, September 2000.

[14] Karl-Filip Faxén. The costs and benefits of cloning in a lazy functional language. In Stephen Gilmore, editor, *Trends in Functional Programming, volume 2*, pages 1–12. Intellect, 2001. Proc. of Scottish Functional Programming Workshop, 2000.

[15] Mark P. Jones. A theory of qualified types. In *European symposium on programming, ESOP '92*, Rennes, France, February 1992. Springer Verlag LNCS 582.

[16] Mark P. Jones. Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, Dept. of Computer Science, Yale University, April 1993.

[17] Robert Metzger and Sean Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.

[18] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, pages 269–281. Springer Verlag, April 1980. LNCS 83.

[19] Simon Peyton Jones, John Hughes, et al. Report on the programming language Haskell 98. Available from `www.haskell.org`, February 1999.

[20] J. Plevyak and A. A. Chien. Type directed cloning for object oriented programs. In *Workshop for Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[21] Gérman Puebla and Manuel Hermenegildo. Abstract specialization and its applications. In *Proc. of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, June 2003.

[22] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.