Optimising Typed Programs

Martin Elsman*
University of Copenhagen

January 6, 1998

Abstract

In this note we present a set of optimisations for an intermediate language of a Standard ML compiler. Most of the optimisations presented are off-the-shelf optimisations, including dead code elimination, constant folding, recursive function specialization, in-lining and value propagation. All optimisations are presented in a typed setting.

1 Introduction

Typed intermediate languages of optimising compilers are becoming increasingly recognised, mainly for two reasons. First, several type based optimisations and analyses have been suggested that cannot be done in an untyped setting. Such analyses and optimisations include various sorts of boxing analyses [Ler92, HJ94], intensional polymorphism [HM95] and region inference [TT94, BTV96]. Second, types can be used to provide certain safety guarantees for a program. In particular, by propagating types all the way to the target language, it becomes possible to type check programs just before execution. This has the advantage that typed executables may be shipped across the internet and if the executable type checks, it can be trusted.

In this note we describe optimisations for an intermediate language in the ML Kit with Regions compiler [TBE+97] (from hereon just the Kit) which is an optimising compiler for Standard ML [MTHM97]. We show that many of the important optimisations that are possible in an untyped setting, are also possible in a typed setting.

Optimisations performed in an intermediate language of an optimising compiler must satisfy at least two conditions. First, the optimisations must preserve types and semantics. Second, the optimisations may not lead to slower programs or programs that take up more space. In particular, it is important that all optimisations are "safe for space complexity." That is, no optimisation may destroy space complexity properties of the program. Since space consumptions depend

*Address: Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, DK–2100 Copenhagen, Denmark; email: mael@diku.dk.

on the underlying implementation technique, different restrictions apply for different underlying implementation techniques. In compilers building on garbage collection techniques one must be careful that no expression that may capture data in a closure is in-lined under a lambda-binding. For instance in-lining a selection from a large tuple in the body of a function may cause dead data to be captured in the closure [App92]. This restriction must also be enforced in a compiler building on region inference. Moreover, region inference and region representation analyses lays further restrictions on what optimisations may be performed. Yet, still quite a large set of optimisations are possible. We will not in this note address this issue further.

The optimisations that we present are off-the-shelf optimisations, including dead code elimination, constant folding, recursive function specialization, inlining and value propagation. Some optimisations trigger other optimisations and vice versa. In the Kit these optimisations are naturally grouped together in what is called a contract phase. By keeping track of usage counts the contract phase may be implemented as a quasi-one-pass algorithm [AJ97]. In the following we focus on a small example language and present the optimisations for this language.

2 The Language and its Semantics

The language that we consider is a typed lambda language including a let-construct for polymorphism and a letrec-construct for expressing recursion. Further, the language includes constructs for records and sums.

We assume TyVar be a denumerably infinite set of *type variables*, ranged over by α. Types and type schemes conform to the following syntax.

$$\begin{array}{lll} \tau & \texttt{::=} & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \texttt{bool} \\ \sigma & \texttt{::=} & \forall\vec{\alpha}.\tau \end{array}$$

Type schemes are considered equal up-to renaming of bound type variables. A *substitution* S is a finite map from type variables to types. When A is any object and S is a substitution we write $S(A)$ to mean simultaneous capture free substitution of S on A.

For any type scheme $\sigma = \forall\alpha_1 \cdots \alpha_n.\tau$ and type τ', we say that τ' is an *instance of σ (via S)*, written $\sigma \geq \tau'$, if there exists a substitution $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that $S(\tau) = \tau'$. The *instance list of S*, written $il(S)$, is the list $[\tau_1, \ldots, \tau_n]$ and we shall refer to lists of the above form as instance lists and use il to range over them. When $\vec{\alpha} = \alpha_1 \cdots \alpha_n$, $n \geq 0$ is a list of type variables and $il = [\tau_1, \cdots, \tau_n]$ is an instance list we write $\{il/\vec{\alpha}\}$ to mean the substitution $\{\alpha_1 \mapsto \tau_1, \cdots, \alpha_n \mapsto \tau_n\}$. Further, when A is any object we denote by $\mathrm{ftv}(A)$ the set of type variables occurring free in A. We consider a type τ to be the type scheme $\forall().\tau$, hence the set of types is a subset of the set of type schemes.

2.1 Typed Expressions

In the following we use f, x and y to range over a denumerably infinite set Lvar of *lambda variables*. The grammar for typed expressions is given below.

$$\begin{array}{rcl} e & ::= & \lambda x:\tau.e \mid e_1\ e_2 \mid (e_1,e_2) \mid \pi_i\ e \mid x_{il} \\ & \mid & \texttt{let}\ x:\sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{true} \mid \texttt{false} \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & \texttt{letrec}\ f:\sigma\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \end{array}$$

We sometimes abbreviate x_{il} with x, when $il = []$.

A *type environment*, Γ, is a mapping from lambda variables to type schemes. The *static semantics* for the language relates types to expressions, under some assumptions, and describes what expressions are well-typed. The static semantics is presented as a set of inference rules allowing inferences among sentences of the form $\Gamma \vdash e : \tau$, where Γ is a type environment, e an expression and τ a type.

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma + \{x \mapsto \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x:\tau.e : \tau \to \tau'} \quad (1) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad (2)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1,e_2) : \tau_1 \times \tau_2} \quad (3) \qquad \frac{i \in \{1,2\} \quad \Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \quad (4)$$

$$\frac{\Gamma(x) \geq \tau \text{ via } S}{\Gamma \vdash x_{il(S)} : \tau} \quad (5) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \vec{\alpha} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma) \quad \Gamma + \{x \mapsto \forall\vec{\alpha}.\tau\} \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let}\ x : \forall\vec{\alpha}.\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau'} \quad (6)$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \quad (7) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \quad (8)$$

$$\frac{\tau = \tau' \to \tau'' \quad \vec{\alpha} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma) \quad \Gamma + \{f \mapsto \tau\} \vdash \lambda x : \tau'.e_1 : \tau \quad \Gamma + \{f \mapsto \forall\vec{\alpha}.\tau\} \vdash e_2 : \tau'''}{\Gamma \vdash \texttt{letrec}\ f : \forall\vec{\alpha}.\tau\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau'''} \quad (9) \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \quad (10)$$

When e is any expression we write $\mathrm{flv}(e)$ to mean the set of free lambda variables in e. Further, when e and e' are expressions and x is a lambda variable, we

write $e\{e'/x\}$ to mean capture free substitution of e' for x in e. Expressions are considered equal up-to renaming of bound lambda variables and type variables.

The *restriction* of an environment Γ to a set of lambda variables $A \subseteq \mathrm{Dom}(\Gamma)$, written $\Gamma \downarrow A$, is the environment with domain A and values $(\Gamma \downarrow A)(x) = \Gamma(x)$. Further, we say that an environment Γ *enriches* another environment Γ', written $\Gamma \sqsupseteq \Gamma'$, if $\mathrm{Dom}(\Gamma) \supseteq \mathrm{Dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathrm{Dom}(\Gamma')$.

The following restriction lemma can be proved by induction over the structure of expressions.

Lemma 2.1 (Elaboration closed under restriction) *For all environments Γ and Γ', expressions e and types τ, if $\Gamma \vdash e : \tau$ and $\Gamma' \sqsupseteq (\Gamma \downarrow \mathrm{flv}(e))$ then $\Gamma' \vdash e : \tau$.*

Further, the following substitution lemma can be proved by induction over the structure of expressions.

Lemma 2.2 (Elaboration closed under substitution) *For all environments Γ, expressions e, types τ and substitutions S, if $\Gamma \vdash e : \tau$ then $S(\Gamma) \vdash S(e) : S(\tau)$.*

2.2 Untyped Expressions

To obtain an untyped expression from a typed expression we define an erasure operation er. A couple of the defining equations are given below.

$$\begin{aligned} er(\lambda x : \tau.e) &= \lambda x.er(e) \\ er(e_1\ e_2) &= er(e_1)\ er(e_2) \\ &\vdots \end{aligned}$$

The *dynamic semantics* of the language relates untyped expressions to so called values, under some assumptions associating values to variables. Thus, a *dynamic environment*, $\mathcal{E}$, maps lambda variables to values, which again conform to the grammar:

$$v \quad \texttt{::=} \quad \texttt{clos}(\lambda x.e, \mathcal{E}) \ \mid\ \texttt{true} \ \mid\ \texttt{false} \ \mid\ (v_1, v_2)$$

The rules of the dynamic semantics allows inferences among sentences of the form $\mathcal{E} \vdash e \Downarrow v$, where $\mathcal{E}$ is a dynamic environment, e is an untyped expression and v is a value. To give meaning to recursive functions we allow creation of non-well-founded objects; i.e. closures, cl_∞, with the property

$$cl_\infty = \texttt{clos}(\lambda x.e, \mathcal{E} + \{f \mapsto cl_\infty\})$$

where x and f are lambda variables, e is an expression and $\mathcal{E}$ is a dynamic environment [MT91].

Expressions $\boxed{\mathcal{E} \vdash e \Downarrow v}$

$$\frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \Downarrow v} \quad (11) \qquad \frac{}{\mathcal{E} \vdash \lambda x.e \Downarrow \texttt{clos}(\lambda x.e, \mathcal{E})} \quad (12)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow \texttt{clos}(\lambda x.e, \mathcal{E}_0) \\ \mathcal{E} \vdash e_2 \Downarrow v \quad \mathcal{E}_0 + \{x \mapsto v\} \vdash e \Downarrow v'\end{array}}{\mathcal{E} \vdash e_1\ e_2 \Downarrow v'} \quad (13)$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow v_1 \quad \mathcal{E} \vdash e_2 \Downarrow v_2}{\mathcal{E} \vdash (e_1, e_2) \Downarrow (v_1, v_2)}\ (14) \qquad \frac{i \in \{1,2\} \quad \mathcal{E} \vdash e \Downarrow (v_1, v_2)}{\mathcal{E} \vdash \pi_i\ e \Downarrow v_i}\ (15)$$

$$\frac{\mathcal{E} \vdash e \Downarrow \texttt{true} \quad \mathcal{E} \vdash e_1 \Downarrow v}{\mathcal{E} \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}\ (16) \qquad \frac{\mathcal{E} \vdash e \Downarrow \texttt{false} \quad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}\ (17)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow v_1 \\ \mathcal{E} + \{x \mapsto v_1\} \vdash e_2 \Downarrow v_2\end{array}}{\mathcal{E} \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v_2} \quad (18) \qquad \frac{}{\mathcal{E} \vdash \texttt{false} \Downarrow \texttt{false}}\ (19)$$

$$\frac{\begin{array}{c}cl_\infty = \texttt{clos}(\lambda x.e_1, \mathcal{E} + \{f \mapsto cl_\infty\}) \\ \mathcal{E} + \{f \mapsto cl_\infty\} \vdash e_2 \Downarrow v_2\end{array}}{\mathcal{E} \vdash \texttt{letrec}\ f\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v_2}\ (20) \qquad \frac{}{\mathcal{E} \vdash \texttt{true} \Downarrow \texttt{true}} \quad (21)$$

2.3 Expression Contexts

We define three notions of expression contexts; local expression contexts, allowing one to single out a local expression (not going under a lambda or a letrec construct); global expression contexts, allowing one to single out any subexpression; and multi expression contexts, allowing one to single out multiple occurrences of an expression. A *local expression context*, L, takes the following form.

$$\begin{array}{rcl} L & ::= & [\cdot] \mid L\ e \mid e\ L \mid (L, e) \mid (e, L) \mid \pi_i\ L \\ & \mid & \texttt{let}\ x : \sigma\ \texttt{=}\ L\ \texttt{in}\ e \mid \texttt{let}\ x : \sigma\ \texttt{=}\ e\ \texttt{in}\ L \\ & \mid & \texttt{letrec}\ f : \sigma\ x\ \texttt{=}\ e\ \texttt{in}\ L \\ & \mid & \texttt{if}\ L\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & \texttt{if}\ e\ \texttt{then}\ L\ \texttt{else}\ e_2 \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ L \end{array}$$

Further, *global expression contexts*, C, takes the following form.

$$C \quad ::= \quad L \mid L[\lambda x : \tau.\ C] \mid L[\texttt{letrec}\ f : \sigma\ x = C\ \texttt{in}\ e]$$

Finally, *multi expression contexts*, M, takes the following form.

$$\begin{array}{rcl} M & ::= & [\cdot] \mid e \mid \lambda x : \tau.M \mid M_1\ M_2 \mid (M_1, M_2) \mid \pi_i\ M \\ & \mid & \texttt{let}\ x : \sigma = M_1\ \texttt{in}\ M_2 \mid \texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_2 \\ & \mid & \texttt{letrec}\ f : \sigma\ x = M_1\ \texttt{in}\ M_2 \end{array}$$

When $\mathcal{C}$ is either a local expression context, a global expression context or a multi expression context, and e is an expression, the effect of *filling* $\mathcal{C}$ with e, written $\mathcal{C}[e]$ is to replace all appearances of the hole in $\mathcal{C}$ with e. We write $\mathrm{flv}(\mathcal{C})$ to mean the set of free lambda variables in $\mathcal{C}$.

2.4 Small, Calling and Safe Expressions

A *small* expression is an expression with at most k_{small} nodes in the abstract syntax tree for the expression. The constant controls what non-recursive functions are in-lined and what recursive functions are specialized. It is a trade-off between code-size and speed. In the Kit a value of 20 is used.

An expression e is considered a *calling expression* if it can be written, $L[e_1\ e_2]$, where L is any local expression context and e_1 and e_2 are any expressions.

The set of terminating expressions that cannot perform any side-effects on the store and cannot raise any exceptions are candidates to dead code elimination and other kinds of optimising reductions. As a simple approximation we consider an expression to be safe if it is not a calling expression and if it cannot raise any exception or update the store. In the small example language considered here an expression is *safe* if it is not a calling expression.

3 Simple Optimising Reductions

In the following we use the term reduction to refer to optimisation reductions, relating expressions. We use $\longrightarrow$ to range over reductions, and we say that e *reduces* to e' if $e \longrightarrow e'$. We now define what it means for a reduction to be type preserving.

Definition 3.1 A reduction, $\longrightarrow$, is *type preserving*, if for all environments Γ, expressions e and e', and types τ, such that $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$. □

The following lemma can be proved by induction over the structure of global expression contexts.

Lemma 3.2 *Assume $\longrightarrow$ is type preserving. For all environments Γ, expressions e and e', global expression contexts C and types τ, if $\Gamma \vdash C[e] : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash C[e'] : \tau$.*

We now present a set of simple optimising reductions. By use of Lemma 2.1 and Lemma 2.2 each of the reductions can be shown to preserve types.

3.1 Reduction of Projections from Explicit Records

Folding of conditional expressions on constants is implemented by value propagation. Reduction of projections from explicit records is implemented by the following rules.

$$\pi_1\ (e_1, e_2) \quad \overset{e_2 \text{ safe}}{\longrightarrow}_{\text{proj}} \quad e_1 \tag{22}$$

$$\pi_2\ (e_1, e_2) \quad \overset{e_1 \text{ safe}}{\longrightarrow}_{\text{proj}} \quad e_2 \tag{23}$$

3.2 Dead Code Elimination

Dead code elimination is implemented by the following rules.

$$\texttt{let}\ x : \sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \quad \overset{\substack{e_1 \text{ safe} \\ x \notin \text{flv}(e_2)}}{\longrightarrow}_{\text{dce}} \quad e_2 \tag{24}$$

$$\texttt{letrec}\ f : \sigma\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \quad \overset{f \notin \text{flv}(e_2)}{\longrightarrow}_{\text{dce}} \quad e_2 \tag{25}$$

3.3 Let-reductions

Simple let-constructs are reduced and explicit applications of lambda-constructs are reduced to let-constructs.

$$\texttt{let}\ x : \forall\vec{\alpha}.\tau\ \texttt{=}\ e_1\ \texttt{in}\ x_{il} \quad \longrightarrow_{\text{let}} \quad e_1\{il/\vec{\alpha}\} \tag{26}$$

$$(\lambda x : \tau.e_2)\ e_1 \quad \longrightarrow_{\lambda} \quad \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \tag{27}$$

3.4 Letrec-reductions

Several reductions are performed on letrec-constructs. Non-recursive functions bound by letrec-constructs are reduced to let-bindings of lambda-constructs as follows.

$$\begin{aligned}\texttt{letrec}\ f : \forall\vec{\alpha}.\tau_1 \to \tau_2\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \quad &\overset{f \notin \text{flv}(e_1)}{\longrightarrow}_{\text{rec1}} \\ \texttt{let}\ f : \forall\vec{\alpha}.\tau_1 \to \tau_2\ \texttt{=}\ \lambda x : \tau_1.e_1\ \texttt{in}\ e_2 &\end{aligned} \tag{28}$$

The following reduction rule opens for other reductions.

$$\begin{aligned}(\texttt{letrec}\ f : \sigma\ x\ \texttt{=}\ e_1\ \texttt{in}\ f_{il})\ e_2 \quad &\longrightarrow_{\text{rec2}} \\ \texttt{letrec}\ f : \sigma\ x\ \texttt{=}\ e_1\ \texttt{in}\ (f_{il}\ e_2) &\end{aligned} \tag{29}$$

4 In-lining Non-recursive Functions

In-lining (β-reduction) is an important optimisation for functional programs. Non-recursive functions referenced only once may always be in-lined and small functions referenced more than once may also be in-lined. The Kit implements the following in-lining strategies.

$$\texttt{let}\ f : \forall\vec{\alpha}.\tau\ \texttt{=}\ \lambda x : \tau'.e\ \texttt{in}\ C[f_{il}] \quad \overset{f \notin \mathrm{flv}(C)}{\longrightarrow}_{\mathrm{inl1}} \quad C[(\lambda x : \tau'.e)\{il/\vec{\alpha}\}] \tag{30}$$

$$\texttt{let}\ f : \forall\vec{\alpha}.\tau\ \texttt{=}\ \lambda x : \tau'.e\ \texttt{in}\ e' \quad \overset{e\ \mathrm{small}}{\longrightarrow}_{\mathrm{inl2}} \quad e'\{((\lambda x : \tau'.e)\{il/\vec{\alpha}\})/f_{il}\} \tag{31}$$

5 Specializing Recursive Functions

Specialization of recursive functions is an important optimisation in a compiler for a functional language [SW95]. In particular specializations of small functions as *fold* and *map* with respect to their first arguments lead to important speedups without drastically increasing the code size.

As an example consider the following Standard ML program.

```
let fun map f [] = []
      | map f (x::xs) = f x :: map f xs
in map (fn x => x+1) [1,2]
end
```

By specializing recursive functions this program is transformed into the following optimised program.

```
let fun map [] = []
      | map (x::xs) = x+1 :: map xs
in map [1,2]
end
```

The function `map` is now first-order and further, the successor function is in-lined into the body of the `map` function. This optimisation has an important effect on performance.

Small recursive functions may be specialized according to the following rule.

$$\begin{array}{l}\texttt{letrec}\ f : \forall\vec{\alpha}.\tau_1 \to (\tau_2 \to \tau_3)\ x\ \texttt{=} \\ \quad \lambda y : \tau_2.M[f\ x] \\ \texttt{in}\ C[f_{il}\ e]\end{array} \quad \overset{\substack{f \notin \mathrm{flv}(M) \\ M[f]\ \mathrm{small}}}{\longrightarrow}_{\mathrm{spec1}} \tag{32}$$

$$\begin{array}{l}\texttt{letrec}\ f : \forall\vec{\alpha}.\tau_1 \to (\tau_2 \to \tau_3)\ x\ \texttt{=} \\ \quad \lambda y : \tau_2.M[f\ x] \\ \texttt{in}\ C\left[\begin{array}{l}\texttt{let}\ x : \tau_1\{il/\vec{\alpha}\}\ \texttt{=}\ e\ \texttt{in} \\ \texttt{letrec}\ f : (\tau_2 \to \tau_3)\{il/\vec{\alpha}\}\ y\ \texttt{=} \\ \quad M[f]\{il/\vec{\alpha}\} \\ \texttt{in}\ f\end{array}\right]\end{array}$$

Note that the original binding of the recursive function is not removed by the reduction.

Even large recursive functions may be specialized. This is captured by the following rule.

$$\begin{array}{l}\texttt{letrec } f : \forall\vec{\alpha}.\tau_1 \to (\tau_2 \to \tau_3)\ x = \\ \quad \lambda y : \tau_2.M[f\ x] \\ \texttt{in } C[f_{il}\ e]\end{array} \quad \begin{array}{c} f \notin \mathrm{flv}(M) \\ f \notin \mathrm{flv}(C) \cup \mathrm{flv}(e) \\ \longrightarrow_{\mathrm{spec2}} \end{array} \quad (33)$$

$$C\left[\begin{array}{l}\texttt{let } x : \tau_1\{il/\vec{\alpha}\} = e \texttt{ in} \\ \texttt{letrec } f : (\tau_2 \to \tau_3)\{il/\vec{\alpha}\}\ y = \\ \quad M[f]\{il/\vec{\alpha}\} \\ \texttt{in } f\end{array}\right]$$

In this case the original binding of the recursive function is removed by the reduction.

6 Value Propagation

By propagating information throughout a program about to what kinds of values a variable is bound, it is possible to eliminate many unnecessary fetches from records and unnecessary conditional checks. The set of abstract values AbsVal is defined by the following syntax. We use V to range over AbsVal.

$$V ::= \top \mid x_{il} \mid (V_1, V_2) \mid \texttt{true} \mid \texttt{false}$$

We say that an abstract value V is *simple* if V is either a constant `true` or `false`, or a variable x_{il}. When V_1 and V_2 are abstract values we define the *least upper bound* of V_1 and V_2, written $V_1 \sqcap V_2$, recursively as follows.

$$V \sqcap V' \quad = \quad \begin{cases} V & \text{if } V = V' \text{ and } V \text{ simple} \\ (V_1 \sqcap V_1', V_2 \sqcap V_2') & \text{if } V = (V_1, V_2) \text{ and } V' = (V_1', V_2') \\ \top & \text{otherwise} \end{cases}$$

Further, when V is an abstract value and x is a lambda variable, we define the *exclusion* of x from V, written $V \setminus\!\setminus x$, recursively as follows.

$$V \setminus\!\setminus x \quad = \quad \begin{cases} \top & \text{if } V = x_{il} \\ (V_1 \setminus\!\setminus x, V_2 \setminus\!\setminus x) & \text{if } V = (V_1, V_2) \\ V & \text{otherwise} \end{cases}$$

A propagation environment Φ is a finite mapping from lambda variables to pairs of a list of type variables and an abstract value.

$$\Phi \quad \in \quad \mathrm{PropEnv} = \mathrm{Lvar} \xrightarrow{\mathrm{fin}} \mathrm{TyVar}^{(k)} \times \mathrm{AbsVal}$$

Below we state a propagation function $\mathcal{P} : \mathrm{Lexp} \to \mathrm{PropEnv} \to \mathrm{Lexp} \times \mathrm{AbsVal}$. Given an expression and a propagation environment the propagation function $\mathcal{P}$ computes an optimised expression and an abstract value for the expression.

$$
\begin{array}{lcl}
\mathcal{P}\ [\![\texttt{true}]\!]\ \Phi & = & (\texttt{true}, \texttt{true}) \\
\mathcal{P}\ [\![\texttt{false}]\!]\ \Phi & = & (\texttt{false}, \texttt{false})
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}\ [\![\lambda x : \tau . e]\!]\ \Phi & = & \text{let } (e',\ \_) = \mathcal{P}\ [\![e]\!]\ (\Phi + \{x \mapsto ([], \top)\}) \\
 & & \text{in } (\lambda x : \tau . e',\ \top)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}\ [\![e_1\ e_2]\!]\ \Phi & = & \text{let } (e_1',\ \_) = \mathcal{P}\ [\![e_1]\!]\ \Phi \\
 & & \quad\ (e_2',\ \_) = \mathcal{P}\ [\![e_2]\!]\ \Phi \\
 & & \text{in } (e_1'\ e_2',\ \top)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}\ [\![(e_1, e_2)]\!]\ \Phi & = & \text{let } (e_1', V_1) = \mathcal{P}\ [\![e_1]\!]\ \Phi \\
 & & \quad\ (e_2', V_2) = \mathcal{P}\ [\![e_2]\!]\ \Phi \\
 & & \text{in } ((e_1', e_2'),\ (V_1, V_2))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}\ [\![\texttt{let}\ x : \forall \vec{\alpha} . \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\ \Phi \quad = \\
\quad \text{let } (e_1',\ V_1) = \mathcal{P}\ [\![e_1]\!]\ \Phi \\
\quad\quad\ (e_2',\ V_2) = \mathcal{P}\ [\![e_2]\!]\ (\Phi + \{x \mapsto (\vec{\alpha}, V_1)\}) \\
\quad \text{in } (\texttt{let}\ x : \forall \vec{\alpha} . \tau\ \texttt{=}\ e_1'\ \texttt{in}\ e_2',\ V_2 \setminus\!\setminus x)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}\ [\![\texttt{letrec}\ f : \forall \vec{\alpha} . \tau\ x\ \texttt{=}\ e_1 \texttt{in}\ e_2]\!]\ \Phi \quad = \\
\quad \text{let } (e_1',\ \_) = \mathcal{P}\ [\![e_1]\!]\ (\Phi + \{f \mapsto ([], \top), x \mapsto ([], \top)\}) \\
\quad\quad\ (e_2',\ V_2) = \mathcal{P}\ [\![e_2]\!]\ (\Phi + \{f \mapsto (\vec{\alpha}, \top)\}) \\
\quad \text{in } (\texttt{letrec}\ f : \forall \vec{\alpha} . \tau\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2',\ V_2 \setminus\!\setminus f)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}\ [\![\pi_i\ e]\!]\ \Phi \quad = \\
\quad \begin{cases}
(V_i,\ V_i) & \text{if } V = (V_1, V_2),\ V_i \text{ simple and } e' \text{ safe} \\
(\pi_i\ e',\ V_i) & \text{if } V = (V_1, V_2) \text{ and } (V_i \text{ not simple or } e' \text{ not safe}) \\
(\pi_i\ e',\ \top) & \text{otherwise}
\end{cases} \\
\quad \text{where } (e',\ V) = \mathcal{P}\ [\![e]\!]\ \Phi
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}\ [\![\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\ \Phi \quad = \\
\quad \begin{cases}
(e_1',\ V_1) & \text{if } V' \text{ not simple},\ V = \texttt{true} \text{ and } e' \text{ safe} \\
(e_2',\ V_2) & \text{if } V' \text{ not simple},\ V = \texttt{false} \text{ and } e' \text{ safe} \\
(V',\ V') & \text{if } V' \text{ simple and } e' \text{ safe} \\
(\texttt{if}\ e'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2',\ V') & \text{otherwise}
\end{cases} \\
\quad \text{where } (e',\ V) = \mathcal{P}\ [\![e]\!]\ \Phi \\
\qquad\quad (\Phi_{\mathrm{t}}, \Phi_{\mathrm{f}}) = \text{if } V = x_{[]} \text{ then } (\ \Phi + \{x \mapsto ([], \texttt{true})\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \Phi + \{x \mapsto ([], \texttt{false})\}) \\
\qquad\qquad\qquad\qquad\quad \text{else } (\Phi, \Phi) \\
\qquad\quad (e_1',\ V_1) = \mathcal{P}\ [\![e_1]\!]\ \Phi_{\mathrm{t}} \\
\qquad\quad (e_2',\ V_2) = \mathcal{P}\ [\![e_2]\!]\ \Phi_{\mathrm{f}} \\
\qquad\quad V' = V_1 \sqcap V_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}\ [\![x_{il}]\!]\ \Phi & = & \begin{cases}
(V,\ V)\{il/\vec{\alpha}\} & \text{if } V \text{ simple} \\
(x_{il},\ x_{il}) & \text{if } V = \top \\
(x_{il}, V\{il/\vec{\alpha}\}) & \text{otherwise}
\end{cases} \\
 & & \text{where } (\vec{\alpha},\ V) = \Phi(x)
\end{array}
$$

7 Implementation

All optimisations presented here have been implemented in the ML Kit with Regions compiler. Due to the sensitivity of region inference and region representation analyses w.r.t. changes in the program, the value propagation algorithm used in the Kit is not as aggressive as the algorithm presented here.

A straight-forward implementation of the optimisation rules will quickly show to be at least quadratic in the size of the program. Inspired by [AJ97] the optimisations presented here are implemented in the Kit by two functions, *reduce* and *contract*. The function *contract* maintains an environment and applies the function *reduce* on the way down the syntax tree and the way up the syntax tree. The function *reduce* reduces redexes w.r.t. environment and usage information. It updates usage information when inserting and removing sub-expressions.

Consider the expression `let` x `=` e_1 `in` e_2. On the way down, if there is zero uses of x then eliminate e_1 (if it is safe) and decrement uses in e_1 prior to recurring on e_2. This may trigger in-lining in e_2 of functions also applied in e_1, e.g. On the way up, if there is now zero uses of x then eliminate the binding (if it is safe) and decrement uses in e_1 prior to returning.

8 Conclusions

In this note we have presented a set of off-the-shelf optimisations for a typed intermediate language of the Kit which is a Standard ML compiler. It has not been possible to present all optimisations performed in the Kit using the small example language presented in this note. For instance, the intermediate language of the Kit includes a fix-construct to allow for mutually recursive functions. As an optimisation, the Kit minimises each fix-construct by finding strongly connected components of the call-graph associated with each fix-construct.

Further, the Kit implements a few other optimisations that we do not mention here. These are optimisations that are performed only to improve on region inference.

References

[AJ97] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. In *Journal of Functional Programming*, 1997.

[App92] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23st ACM Symposium on Principles of Programming Languages*, January 1996.

[HJ94] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Portland, Oregon*, pages 213–226, January 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages*, San Francisco, January 1995.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Principles of Programming Languages*, pages 177–188, 1992.

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. In *Theoretical Computer Science 87*, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Second International Symposium on Static Analysis (SAS)*, pages 366–381, September 1995.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Dept. of Computer Science, University of Copenhagen, 1997. (http://www.diku.dk/research-groups/topps/activities/kit2).

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages*, January 1994.