Strictness Detection in Non-Flat Domains

John Hughes,
Institutionen før Informationsbehandling,
Chalmers Tekniska Høgskola,
41296 Gøteborg,
SWEDEN.

From January 1st 1986:
Department of Computer Science,
University of Glasgow,
University Avenue,
Glasgow,
UNITED KINGDOM.

Introduction

A function f is said to be *strict* if f $\perp$=$\perp$, that is, if whenever the evaluation of its argument fails to terminate then so does its result. Knowledge about strictness is of great practical importance in the implementation of functional languages, because of the freedom it gives to change the evaluation order. For example, the argument of a strict function can safely be evaluated before the call, avoiding the overhead normally associated with lazy evaluation. This is done in the Ponder compiler, where it appears to improve performance by a factor of between three and five[3]. Alternatively, the arguments of strict functions can be evaluated in parallel with the call, leading to better exploitation of parallel hardware[2].

In order to enjoy these benefits it is necessary to be able to identify strict functions automatically. The classic paper on this subject is Mycroft's[5]. Mycroft gives a method which can identify very many strict functions, but which has serious limitations - it is restricted to first order functions on flat domains. Since list processing and higher order functions are very common in functional languages the benefits of strictness analysis are much reduced. A Ponder implementation of Quicksort, for example, gains only 10% from strictness analysis. Recently Wray[6], Hudak[4] and Burn et al.[1] have extended Mycroft's method to deal with higher-order functions. In this paper we give a method which works for functions on non-flat domains.

The Source Language

The language we analyse is a very simple first order functional language. A program is a set of equations, each defining a function. For example, `append` and `reverse` are defined as follows:

```
append a b = case a:
                 nil → b
                 cons c d → cons c (append d b)

reverse a =  case a:
                 nil → nil
                 cons b c → append (reverse c) (cons b nil)
```

These examples use all the constructs of the language. There are three kinds of expressions: variables, function applications, and **case** expressions. Data structures are built using constructors such as `cons` and `nil`, and taken apart by **case** expressions. The patterns in a **case** expression consist of a constructor and variable names which are bound to the components. Since we are only interested in analysing the strictness of functions on data structures we have not included any "built-in" types such as integers or booleans. We call this language *FAD* (Functions and Data-structures).

The domain of *FAD* values, **V**, is the solution of the domain equation

$$\mathbf{V} = \mathbf{CON} \otimes \mathbf{V}*$$

where **CON** is a flat domain of constructors and $\otimes$ is the coalesced product. Every non-$\perp$ value has a non-$\perp$ constructor and a non-$\perp$ sequence of components, although any component may be $\perp$. (We ignore the arity of constructors in this equation). Given this domain, the denotational semantics of *FAD* is obvious and we will say no more about it.

Representing Lazy Closures

In any implementation of a lazy language values may be represented as *closures*. A closure contains all the information necessary to compute the associated value, and this is done when the value is actually required. In the standard semantics the distinction between closures and values is invisible. We introduce a domain **R** of representations, which can distinguish closures from the values they represent. We define **R** by

$$\mathbf{R} = \mathbf{2} \times (\mathbf{CON} \otimes \mathbf{R}^\circ)$$

where **2** is the two-point domain $\{0 \sqsubseteq 1\}$ and $\mathbf{R}^\circ$ is the domain of sequences with non-$\perp$ elements. A closure is represented by a pair whose first component is 1, and a value is represented by a pair whose first component is 0. The function which evaluates a closure is `eval`, defined by

```
eval: R → R
eval <flag, v> = <0, v>
```

(note that the components of the result may still be closures). The intuition behind the definition of **R** is that a closure may represent $\perp$ while being perfectly well-defined itself. The closure

which evaluates to $\perp$ is represented in **R** by <1, $\perp$>, which is indeed not $\perp$. Note that the components of data-structures may themselves be closures, and that any undefined component *must* be represented by a closure, otherwise the data-structure it is part of is also undefined.

V can be embedded in **R** by the function

```
embed: V → R
embed <con, <v1, .., vn>> = <1, <con, <embed v1, .., embed vn>>>
```

which sets all the "closure flags" to 1. Elements of **V** can be recovered using the function

```
value: R → V
value <flag, <con, <r1, .., rn>>> =
              <con, <value r1, .., value rn>>
```

It is easy to show that

```
value ∘ embed = id
```

We can give an alternative denotational semantics for *FAD* using **R** as the semantic domain. We choose a semantics such that the alternative denotation of any expression is the result of applying embed to its standard denotation. (It follows that no expression denotes $\perp$ in **R**). The alternative semantics is straightforward and we do not give it here.

Since the result of embed is never $\perp$, we can replace all function applications in the alternative semantics by strict function applications, which we write using a colon. `(f:x)` is $\perp$ if x is $\perp$, and `(f x)` otherwise. This models the implementation closely, reflecting the fact that the closure of an argument is constructed before the function is called.

Now suppose that `f` is a strict function in *FAD* (i.e. that `f` <1, $\perp$> = $\perp$ in **R**). An argument to `f` can safely be evaluated before the call. We can model this by replacing calls such as `f:x` by `f:(eval x)`. We can show that, for all closures `x`,

```
f:x = f:(eval x)
```

if `f` is strict, and so the transformation is correct.

`Eval` evaluates the top-level closure only. Depending on the properties of `f`, it might be possible to evaluate some components of the argument in the same way, to potentially unlimited depth. This more complicated pre-evaluation of arguments can be modelled by introducing a more complicated function than `eval`. Our approach to strictness analysis is to find functions that can be introduced in this way, without changing the meaning of the program. We call such functions *contexts*.

Contexts

What kind of function can serve as a context? Intuitively, a context should "do some evaluation", that is, it should map some 1s in its argument to 0s. A convenient way to state this is that a context c should be weaker than id:**R**→**R**. Also, if its argument is already sufficiently well evaluated, a context should have no effect. We may state this as

```
c ∘ c = c
```

We define a context to be a function c:**R**→**R** with these two properties. Contexts form a domain, which allows us to use approximations and limits of contexts.

Let us introduce some simple contexts. One of the simplest is ABSENT, defined by

```
ABSENT <flag, v> = <flag, ⊥>
```

ABSENT can be introduced under the condition that for all closures x

```
f:x = f:(ABSENT x)
    = f:<1, ⊥>
```

That is, f must be a constant function. In other words, an argument's context is ABSENT if the value of the argument is not used by the function.

The ⊥ of the context domain can be interpreted as the "contradictory context". It can be introduced if

```
f:x = f:(⊥ x)
    = f:⊥
    = ⊥
```

that is, if f is never defined. Thus the value of an argument in a context ⊥ is irrelevant: the function applied to it will crash regardless.

Given an n-ary constructor CON and n contexts $c_1, \ldots, c_n$, we can form a context c defined by

```
c <flag, <CON, <r1, .., rn>>> = <0, <CON, <c1 r1, .., cn rn>>>
c <flag, <CON', ...>> = ⊥                    if CON ≠ CON'
```

c evaluates its argument and returns ⊥, unless the argument is constructed by the constructor CON, in which case it evaluates the components with $c_1, \ldots, c_n$. We write c as (CON $c_1 .. c_n$). This kind of context can be used when a function requires a particular kind of object as its argument. For example, head requires a cons as its argument, and

```
head:x = head:((cons id ABSENT) x)
```

Since contexts form a domain, we can also take the least upper bound of two contexts to get a context. We write the least upper bound operator as ⊔. For example,

```
nil ⊔ cons id id
```

is the context that maps values constructed with nil or cons to themselves, and all others to ⊥. This context could be used with a function which fails unless its argument is a list. As another example, the context

```
ABSENT ⊔ cons id id
```

returns a closure which evaluates to ⊥ unless its argument is a cons. This context could be used with a function which is not known to need its argument, but needs a cons if it does.

We will restrict our attention to contexts built from these four primitives: ABSENT, ⊥, constructors, and ⊔. Note that ⊔ distributes over constructors

$$\text{CON } a_1..a_n \sqcup \text{CON } b_1..b_n = \text{CON } (a_1 \sqcup b_1)..(a_n \sqcup b_n)$$

and that ⊥ is a unit for ⊔. We can therefore express every such context as an ⊔ of constructor contexts with distinct constructors, and possibly ABSENT. We say that a context expressed in this way is in *normal form* if all the component contexts are also in normal form. Normal forms (and other context expressions) may well be infinite: we can think of them as infinite trees. We are particularly interested in *finite* normal forms, which can be described by a finite set of equations. Such equations may be thought of as a (potentially cyclic) graph, and can represent rational infinite trees. Such equations are the usual way we describe contexts. Our strictness analyser expresses the contexts it finds as finite normal forms.

As an example of a context in finite normal form, consider spine defined by

```
spine = nil ⊔ cons ABSENT spine
```

Spine maps everything but finite lists to ⊥ (infinite lists are limits of partial lists, which spine maps to ⊥). All the list elements are ignored (mapped to <1, ⊥>) by ABSENT. As an example where spine can be used,

```
length:L = length:(spine L)
```

We say that a context is *strict* if it maps <1, ⊥> to <0, ⊥>. ABSENT is not strict; the other primitive contexts are. It follows that a context in normal form is strict unless it includes ABSENT at the top-level.

Normal forms can be used to guide compilation. An argument in a strict context can be evaluated before the call. An argument in a non-strict, non-ABSENT context must be compiled as a closure: it may or may not be evaluated. An argument in the context ABSENT need not be compiled at all! The code can pass a dummy value (<1, ⊥>) instead: it will never be used.

Strictness Functions

The condition for introducing a context that we gave above was that, for all closures x,

$$f{:}x = f{:}(c\ x)$$

In fact this is stronger than necessary, because it doesn't take into account the way that f's result is used. Suppose (f:x) is itself an argument of a function g, and suppose the context associated with g is c'. Then we only need to insist that

$$(c' \circ f){:}x = (c' \circ f){:}(c\ x)$$

Clearly c depends on c'. We express this dependence by introducing a *strictness function* F such that

$$c = F\ c'$$

satisfies the condition above. By convention we use capitalised versions of function names for the associated strictness functions.

We will show later how a definition of F can be derived from f's own definition, but we can already prove some theorems about strictness functions. For example, we can take F ABSENT = ABSENT for any function f. The condition that F ABSENT must satisfy is

$$(ABSENT \circ f){:}x = (ABSENT \circ f){:}((F\ ABSENT)\ x)$$

which is true provided ((F ABSENT) x) is never $\perp$. The least context with this property is ABSENT.

So far we have discussed functions of one argument. However, *FAD* provides functions of several arguments. If f has n arguments, and a call is in context c', then the condition for using contexts $c_1, \ldots, c_n$ to evaluate the arguments is

$$(c' \circ f){:}x_1{:}\ldots{:}x_n = (c' \circ f){:}(c1\ x_1){:}\ldots{:}(cn\ x_n)$$

(We take the composition of a unary and an n-ary function to be the function that applies the n-ary function to all the n arguments, and applies the unary function to the result). This is equivalent to

$$(c' \circ f){:}x_1{:}\ldots{:}x_n = (c' \circ f){:}x_1{:}\ldots{:}x_{i-1}{:}(c_i\ x_i){:}x_{i+1}{:}\ldots{:}x_n$$

for each i separately. In this case we associate n different strictness functions with f, called $F_1, \ldots, F_n$, such that

$$c_i = F_i\ c'$$

satisfies the condition above.

Operators on Contexts

We will need two other operators on contexts. The first is $\sqcap$, which is used to express the "net context" of an argument that is used several times in different contexts. For example, suppose

$$f\ x = g\ x\ x$$

and for all closures u and v,

$$g{:}u{:}v = g{:}(c_1\ u){:}v = g{:}u{:}(c_2\ v)$$

Then $(c_1 \sqcap c_2)$ is a context c such that

$$g{:}x{:}x = g{:}(c\ x){:}(c\ x)$$

$\sqcap$ can be defined by

$$\begin{aligned}
&\bot \sqcap b = \bot \\
&\mathsf{ABSENT} \sqcap b = b \\
&\mathsf{CON}\ a_1..a_n \sqcap \mathsf{CON}\ b_1..b_n = \mathsf{CON}\ (a_1 \sqcap b_1)..(a_n \sqcap b_n) \\
&\mathsf{CON}\ a_1..a_n \sqcap \mathsf{CON'}\ b_1..b_m = \bot \qquad \text{if CON}\neq\text{CON'} \\
&(a_1 \sqcup a_2) \sqcap b = (a_1 \sqcap b) \sqcup (a_2 \sqcap b)
\end{aligned}$$

together with the commutative and associative laws.

Unfortunately this is the simplest way of characterising $\sqcap$ we have found. The definition can be justified by showing that $(c_1 \sqcap c_2)$ satisfies

$$g{:}x{:}x = g{:}(c\ x){:}(c\ x)$$

in each case. It is tempting to believe that $\sqcap$ is the greatest lower bound operator, but this is not the case. The reason is that, in general, ABSENT $\sqcap$ b is not the greatest lower bound of ABSENT and b. In fact $\sqcap$ and $\sqcup$ do not form a Boolean algebra, despite the suggestive names. However, they are each idempotent, commutative and associative, and they satisfy both distributive laws. They fail to satisfy the cancellation laws

$$\begin{aligned}
&E \sqcup (E \sqcap F) = E \\
&E \sqcap (E \sqcup F) = E
\end{aligned}$$

A counter-example to each of these is E=ABSENT, which satisfies

$$\text{ABSENT} \sqcap F = F$$

but

$$\text{ABSENT} \sqcup F \neq \text{ABSENT}$$

The other operator we will need is arrow $\rightarrow$. We will introduce it by considering an example: the strictness function HEAD associated with head. Let c' be (HEAD c). Then c' must satisfy

$$(c \circ \text{head}) : x = (c \circ \text{head}) : (c'\ x)$$

It is tempting to take c' to be (cons c ABSENT), requiring x to be a cons, applying c to its head and throwing away its tail. However, this is wrong because (cons c ABSENT) is a strict context, and although head is a strict function, (c ∘ head) may not be. For example, if c is ABSENT then c' should also be ABSENT. We need to make the strictness of c' depend on the strictness of c, and this is what arrow does. We define

$$\text{HEAD}\ c = c \rightarrow \text{cons}\ c\ \text{ABSENT}$$

where

$$\begin{aligned}
&\text{ABSENT} \rightarrow b = \text{ABSENT} \\
&\text{strict context} \rightarrow b = b \\
&(a_1 \sqcup a_2) \rightarrow b = (a_1 \rightarrow b) \sqcup (a_2 \rightarrow b)
\end{aligned}$$

and for completeness

$$\bot \rightarrow b = \bot$$

Deriving Definitions of Strictness Functions

Given a function definition

$$f\ a_1\ ..\ a_n = E$$

we want to derive definitions of the associated strictness functions $F_1,..,F_n$. With this aim we introduce a function D such that $D_x[\![E]\!]c$ is a context c' satisfying

$$(c \circ \lambda x.E) : u = (c \circ \lambda x.E) : (c'\ u)$$

for all closures u, and for all values of the free variables of E. Intuitively, c' is the net context applied to x when c is applied to E. The strictness functions can then be defined by

$$F_i\ c = D_{a_i}[\![E]\!]c$$

D is defined by cases. Clearly, if x does not occur free in E then

$$D_x[\![E]\!]c = \text{ABSENT}$$

If E is the identifier x, then

$$D_x[\![x]\!]c = c$$

If E is a function application, then referring to the definition of $\sqcap$ we deduce that

$$D_x[\![f\ E_1\ ..\ E_n]\!]c = D_x[\![E_1]\!](F_1\ c) \sqcap .. \sqcap D_x[\![E_n]\!](F_n\ c)$$

The hardest case is if E is a **case** expression:

case E_0:
$\quad CON_1\ a_{11}\ a_{12}..\ \rightarrow E_1$
$\quad CON_2\ a_{21}..\ \rightarrow E_2$
$\quad ...$
$\quad CON_n\ a_{n1}..\ \rightarrow E_n$

where the CON_i are constructors. First suppose that x does not occur in E_0. Then the selection of one of the E_i is essentially arbitrary (depends on the values of the free variables), and the resulting context c' must be generous enough that whichever E_i is chosen,

$$(c \circ \lambda x.E_i):x = (c \circ \lambda x.E_i):(c'\ x)$$

We can ensure this by taking c' to be the upper bound of the contexts $D_x[\![E_i]\!]c$.

On the other hand, suppose x occurs only in E_0. We can assume that E_0 is precisely x (if not, we introduce $y=E_0$, find $c'=D_y[\![E]\!]c$, and then return $D_x[\![E_0]\!]c'$). Now let α_{ij} be $D_{aij}[\![E_i]\!]c$. Then the context

$$(CON_i\ \alpha_{i1}\ \alpha_{i2}\ ...)$$

evaluates x appropriately if E_i is the case selected. Since we don't know in advance which case will be selected, we take the upper bound of all these contexts. This upper bound is always a strict context, so we make its strictness depend on c using the arrow operator. The result in this case is

$$c \rightarrow (CON_1\ \alpha_{11}\ \alpha_{12}\ ..\ \sqcup\ CON_2\ \alpha_{21}\ ..\ \sqcup\ ..\ \sqcup\ CON_n\ \alpha_{n1}\ ..)$$

In the general case, where x may appear in E_0 and in the E_i we can treat the two kinds of occurrence independently, and then $\sqcap$ the resulting contexts together. The final definition, then, is

$$D_x[\![\textbf{case } E_0\text{:}$$
$$\quad \mathrm{CON}_1\ a_{11}\ a_{12}..\ \rightarrow E_1$$
$$\quad \mathrm{CON}_2\ a_{21}..\ \rightarrow E_2$$
$$\quad ...$$
$$\quad \mathrm{CON}_n\ a_{n1}..\ \rightarrow E_n]\ c =$$

$$(D_x[\![E_1]\!]c \sqcup ... \sqcup D_x[\![E_n]\!]c) \sqcap$$
$$D_x[\![E_0]\!]$$
$$\quad (c \rightarrow \mathrm{CON}_1\ D_{a11}[\![E_1]\!]c.. \sqcup ... \sqcup \mathrm{CON}_n\ D_{an1}[\![E_n]\!]c..)$$

As an example, when we apply these rules to the definitions of head and append, we get the following strictness functions (after a little simplification):

$$\mathrm{HEAD}\ c = c \rightarrow \mathrm{cons}\ c\ \mathrm{ABSENT}$$
$$\mathrm{APPEND}_1\ c = c \rightarrow \mathrm{nil} \sqcup \mathrm{cons}\ (\mathrm{CONS}_1\ c)\ (\mathrm{APPEND}_1\ (\mathrm{CONS}_2\ c))$$
$$\mathrm{APPEND}_2\ c = c \sqcup \mathrm{APPEND}_2\ (\mathrm{CONS}_2\ c)$$

Selectors

The method in the preceding section derives definitions of strictness functions from the definitions in the program. It cannot be used to derive definitions of strictness functions associated with predefined functions. In *FAD* the only predefined functions are constructors, and in this section we define the strictness functions associated with them. All such functions have the same form, which we will illustrate by considering CONS_1.

We define the strictness functions associated with constructors by showing how they act on contexts of various forms. The most interesting case is when the argument context is a matching constructor context, as in the example CONS_1 (cons a b). When the context (cons a b) is applied to the *FAD* expression (cons u v), then the function applied to u is, of course, a. Therefore we can take

$$\mathrm{CONS}_1\ (\mathrm{cons}\ a\ b) = a$$

In the world of contexts, CONS_1 plays an analogous role to head. Similarly CONS_2 plays an analogous role to tail. Because of this analogy we call the strictness functions associated with constructors *selectors*.

The other cases of the definition are straightforward:

$$\mathrm{CONS}_1\ \bot = \bot$$
$$\mathrm{CONS}_1\ \mathrm{ABSENT} = \mathrm{ABSENT}$$
$$\mathrm{CONS}_1\ (\mathrm{CON}\ a_1\ ..\ a_n) = \bot \qquad \text{if CON}\neq\text{cons}$$
$$\mathrm{CONS}_1\ (a \sqcup b) = \mathrm{CONS}_1\ a \sqcup \mathrm{CONS}_1\ b$$

Selectors also satisfy

$$\mathrm{CONS}_1\ (a \sqcap b) = \mathrm{CONS}_1\ a \sqcap \mathrm{CONS}_1\ b$$
$$\mathrm{CONS}_1\ (a \to b) = a \to \mathrm{CONS}_1\ b$$

Analysing Strictness

Now we know how to derive strictness functions, and how to simplify context expressions. We can use this to calculate the extent to which eager evaluation can be used. For example, consider the expression

```
head (append x y)
```

which is itself evaluated by a strict context c. To discover how eagerly x can be evaluated, we compute

$$D_x[\![\mathtt{head(append\ x\ y)}]\!]c = \mathrm{APPEND}_1\ (\mathrm{HEAD}\ c)$$
$$= \mathrm{APPEND}_1\ (c \to \mathtt{cons\ c\ ABSENT})$$
$$= \mathrm{APPEND}_1\ (\mathtt{cons\ c\ ABSENT}) \qquad \text{because c is strict}$$
$$= \mathtt{cons\ c\ ABSENT} \to \mathtt{nil} \sqcup \mathtt{cons}\ (\mathrm{CONS}_1\ (\mathtt{cons\ c\ ABSENT}))\ (\mathrm{APPEND}_1\ (\mathrm{CONS}_2\ (\mathtt{cons\ c\ ABSENT})))$$
$$= \mathtt{nil} \sqcup \mathtt{cons\ c}\ (\mathrm{APPEND}_1\ \mathrm{ABSENT})$$
because (cons c ABSENT) is strict,
CONS_1 (cons c ABSENT) = c,
and CONS_2 (cons c ABSENT) = ABSENT
$$= \mathtt{nil} \sqcup \mathtt{cons\ c\ ABSENT}$$
because F ABSENT = ABSENT for all strictness functions F

which tells us that x can be computed eagerly, that it must yield a nil or a cons, and that if it yields a cons then the tail can be thrown away and the head evaluated by c. In the remainder of the paper we show how these calculations can be done automatically.

It is worth remarking that our approach is inherently less powerful than Mycroft's in the case of a flat domain. This is because we express the strictness of a function of several arguments as a combination of the strictness in each argument independently. Mycroft expresses the strictness of the function as a whole. To see the effect of this, consider the function if, defined by

```
if a b c = case a: true → b, false → c
```

Mycroft's method expresses the fact that (if a b c) evaluates a and b or c, whereas ours only expresses the fact that a is evaluated, and b or c might be. Mycroft's algorithm can therefore conclude that (if a x x) is certain to evaluate x, and ours cannot. Thanks to this

Mycroft can treat conditional as just another function, while a special treatment of the **case** construct is vital to our approach. The advantage of treating arguments independently is that it is simpler, and it is this simplification that has allowed us to extend the method to deal with data structures. In practice one can get the best of both worlds by running both strictness detection algorithms, one after another. It would be best to run Mycroft's algorithm second, as it can take advantage of information already found by our algorithm.

Calculating Contexts

We must now show how to calculate context expressions automatically. We start from a set of equations defining names for the context expressions to be evaluated. We aim to finish with finite normal forms for these variables - that is, another set of equations in a simpler form. We solve the equations in stages, eliminating one kind of operator at each stage, until a finite normal form remains. Note that "weaker" contexts provide more specific information, so it is the *least* solution of the equations that we want to find. However, often this least solution has no finite normal form, and so we look for an *upper bound* on the least solution instead. An upper bound can safely be used by the compiler instead of the least solution - it just carries less information. For example, if the compiler uses an upper bound (ABSENT OR c) instead of a strict context c, the only consequence is that a parameter will be passed as a closure that could have been passed in its evaluated form.

A trick we will use more than once is the *merging* of distinct variables or expressions. Suppose we have the equations

$$a = E_1$$
$$b = E_2$$
$$c = E_3$$
$$\ldots$$

where a, b, c.. are free in $E_1..E_n$. We merge a and b by changing the equations to

$$a = b = E_1 \sqcup E_2$$
$$c = E_3$$
$$\ldots$$

The reason for doing this is that the latter equations may be soluble, even if the former ones are not. For example, if we have an equation of the form

$$a = F\ (G\ a)$$

which we cannot solve, but we know how to solve

$$b = F\ b \sqcup G\ b$$

then we merge a and (G a) to transform the former into the latter.

Of course, this merging changes the solution of the equations in general. Before we use it we must prove that the least solution of the transformed equations is an upper bound for the least solution of the original ones. To see this, let us define G and H by

$$G(\langle a, b, c..\rangle) = \langle E_1, E_2, E_3, ..\rangle$$
$$H(\langle a, b, c..\rangle) = \langle E_1 \sqcup E_2,\ E_1 \sqcup E_2,\ E_3,\ E_4,\ ..\rangle$$

(where a, b, c.. are formal parameters and E_1,E_2,.. are the expressions involving a,b,c.. in the equations above. < and > are tuple brackets). The least solution of the original set of equations is **fix** G, and of the transformed equations is **fix** H. But for all x, G x $\sqsubseteq$ H x, and so G $\sqsubseteq$ H, from which it follows that **fix** G $\sqsubseteq$ **fix** H (here we use $\sqsubseteq$ to denote the domain approximation relation). Therefore the transformation is correct.

Eliminating Strictness Functions

The best way to eliminate calls of strictness functions other then selectors is to replace them with the function bodies, which are exactly equivalent. This is always possible with non-recursive functions, and sometimes possible even with recursive ones. A call of a recursive strictness function can be expanded in this way if there are only finitely many different arguments to all the recursive calls. For example, suppose F is defined by

```
F x = G (F (H x))
```

and H a = b, H b = a. Then c = F a can be expanded as follows:

```
c = F a
  = G (F (H a))
  = G (F b)
  = G (G (F (H b)))
  = G (G (F a))
  = G (G c))
```

Not all recursive functions can be expanded in this way, and in order to avoid trial and error we would like a sufficient condition for the expansion to be possible. Such a condition is that the argument be reduced to finite normal form before the call is expanded, and that the arguments of recursive calls be derived from the top-level argument by applying zero or more selectors. To see this, consider applying a selector to a finite normal form such as

```
spine = nil ⊔ cons ABSENT spine
```

For example, $CONS_2$ spine is

$$\begin{aligned} CONS_2\ spine &= CONS_2\ (nil \sqcup cons\ ABSENT\ spine) \\ &= CONS_2\ nil \sqcup CONS_2\ (cons\ ABSENT\ spine) \\ &\qquad\qquad \text{selectors distribute over } \sqcup \\ &= \bot \sqcup spine \\ &= spine \end{aligned}$$

After distributing the selector over $\sqcup$, all but one of the resulting expressions must simplify to $\bot$, since all the constructors $\sqcup$ed together in a finite normal form are different. Since $\bot$ is an identity for $\sqcup$, all these expressions "drop out", leaving another part of the finite normal form as the result. Since a finite normal form only has a finite number of parts altogether, applying any number of selectors to a finite normal form can give only finitely many different results. If all the arguments to recursive calls of a strictness function are applications of selectors to a finite normal form, then we are certain to be able to perform a finite expansion. (This is a slight oversimplification. Expressions such as ABSENT $\sqcup$ ... in the finite normal form may lead to selectors returning ABSENT $\sqcup$ (an existing expression), but the argument still applies).

As an example, consider

$$\begin{aligned} &APPEND_1\ c = c \rightarrow nil \sqcup cons\ (CONS_1\ c)\ (APPEND_1\ (CONS_2\ c)) \\ &spine = nil \sqcup cons\ ABSENT\ spine \\ &ans = APPEND_1\ spine \end{aligned}$$

The sufficient condition is satisfied: spine is in finite normal form and the argument of the recursive call of $APPEND_1$ is ($CONS_2$ c) - an application of a selector to the top-level argument. In fact $CONS_2$ spine = spine, and so the expansion gives

$$\begin{aligned} ans &= spine \rightarrow nil \sqcup cons\ (CONS_1\ spine)\ (APPEND_1\ spine) \\ &= spine \rightarrow nil \sqcup cons\ (CONS_1\ spine)\ ans \end{aligned}$$

This expansion is exactly equivalent to the original call. When such an expansion cannot be applied we approximate instead, by merging the top-level argument and the arguments of all recursive calls, allowing the results of all calls to be replaced by a single variable. In the case of $APPEND_1$, we start from

$$APPEND_1\ c = c \rightarrow nil \sqcup cons\ (CONS_1\ c)\ (APPEND_1\ (CONS_2\ c))$$

Let a be the argument of the top-level call, and b the argument of the recursive one. Then

$$\begin{aligned} &a = c \\ &b = CONS_2\ a \\ &APPEND_1\ a = a \rightarrow nil \sqcup cons\ (CONS_1\ a)\ (APPEND_1\ b) \end{aligned}$$

Merging a and b gives

$$a = b = c \sqcup CONS_2\ a$$

and now we can replace both calls of $APPEND_1$ by the same variable, g say, giving

$$a = b = c \sqcup CONS_2\ a$$
$$g = a \rightarrow nil \sqcup cons\ (CONS_1\ a)\ g$$

A similar transformation can be applied to mutually recursive functions.

Eliminating Selectors

The next stage is to eliminate selectors (such as $CONS_1$ and $CONS_2$) from the equations. This is by far the hardest part of the solution. Recursive equations such as

$$a = F\ (CONS_2\ a)$$

are often impossible to solve exactly (at least by our methods). In order to avoid too much approximation, and speed up the solution, we begin by splitting the equations into the smallest possible groups with no inter-group recursion (the strongly connected components of the variable dependency graph). We solve the groups one by one in a bottom up order (i.e. we don't solve a group until all groups it depends on have been solved). This guarantees that all the free variables of a group of equations are in finite normal form by the time the group is solved, which allows us to apply selectors to free variables with no difficulties.

We begin our explanation of selector elimination by noting that all selectors satisfy

$$S\ \bot = \bot$$
$$S\ ABSENT = ABSENT$$
$$S\ (a \sqcup b) = S\ a \sqcup S\ b$$
$$S\ (a \sqcap b) = S\ a \sqcap S\ b$$
$$S\ (a \rightarrow b) = a \rightarrow S\ b$$

Moreover, these laws are also satisfied by arbitrary compositions of selectors (including the identity function). We will refer to such compositions as *compound selectors*. The laws can be used to push selectors down through expressions towards the leaves, so that they are only applied to variables and constructions. Furthermore, applications of selectors to constructions can be simplified, giving one of the components of the construction if the selector and constructor match, and $\bot$ if they don't. Applications of selectors to free variables of the whole equation group can be simplified because the free variables are in finite normal form.

Now we take each equation in the group being solved and push the selectors down to the leaves. We introduce a new function F by abstracting out variables in the set being solved, together with any selectors applied to them. We can then express the equation as

$$a = F\ (S_1\ b_1)\ (S_2\ b_2)\ \ldots\ (S_n\ b_n)$$

where the b_i are variables and each S_i is a possibly compound selector (which could therefore be the identity function). F does not involve any selectors, and we can arrange that its parameters are used only once in its body (if a parameter were used twice we could just replace it by two parameters).

Now consider S(F x y z...) where S is a selector and the x, y.. are any expressions. S can be pushed through the body of F, giving

$$S\ (F\ x\ y\ z\ldots) = F'\ x'\ y'\ z'\ldots$$

where F' is another function not involving selectors and where x' is either x or a selector applied to x, y' is either y or a selector applied to y, and so on. (This selector is always S if S is a simple selector, but may be a "prefix" of S if S is compound). For example, if

$$F\ x\ y = x \sqcup y$$

then

$$S\ (F\ x\ y) = S\ (x \sqcup y) = S\ x \sqcup S\ y = F'\ (S\ x)\ (S\ y)$$

with F' actually equal to F. On the other hand, if

$$F\ x\ y = (\text{cons ABSENT } x) \sqcup y$$

then

$$\begin{aligned} CONS_2\ (F\ x\ y) &= CONS_2\ (\text{cons ABSENT } x \sqcup y) \\ &= CONS_2\ (\text{cons ABSENT } x) \sqcup CONS_2\ y \\ &= x \sqcup CONS_2\ y \\ &= F'\ x\ (CONS_2\ y) \end{aligned}$$

It is also possible that one of the parameters disappears completely, as in

$$\begin{aligned} CONS_1\ (F\ x\ y) &= CONS_1\ (\text{cons ABSENT } x \sqcup y) \\ &= \text{ABSENT} \sqcup CONS_1\ y \\ &= F'\ (CONS_1\ y) \end{aligned}$$

but this causes no complications so we ignore it.

We denote the new function F' by S^F. We can show that, for any function F, there are only finitely many different functions S^F, even when S may be any compound selector. The proof is by structural induction on the body of F. We will only sketch the proof here, as it is very easy. The only hard cases are variables. A free variable refers to a context in finite normal form, and we have shown above that applying arbitrary compound selectors to a finite normal form gives only finitely many different results. An application of a selector to a parameter, on the other hand, cannot be simplified, and so results in an expression (S x), which is then abstracted out when S^F is defined. Alpha conversions apart, the particular selector S makes no difference to

the result.

Now let us express the equations to be solved as

$$a_1 = F_1\ (S_{11}\ b_{11})\ (S_{12}\ b_{12})\ldots$$
$$a_2 = F_2\ (S_{21}\ b_{21})\ \ldots$$
$$\ldots$$
$$a_n = F_n\ (S_{n1}\ b_{n1})\ \ldots$$

We have just shown that the set

$$\mathbf{F}^* = \{S\hat{}F \mid F \text{ is one of the } F_i,\ S \text{ is a compound selector}\}$$

is finite. We can therefore calculate it by forming the closure of $\{F_1, .., F_n\}$ under the simple selectors occuring in the equations being solved. Every selector S used in the equations induces a function from $\mathbf{F}^*$ to $\mathbf{F}^*$, which we call S^. S^ is a finite function, and we can calculate it for each selector.

Each variable a_i can be expressed as an infinite expression involving only the $F_1..F_n$ and selectors, just by expanding out the equations we gave above. But now the selectors can be propagated downwards through the tree, using the property

$$S\ (F\ x\ y\ z\ldots) = (S\hat{}F)\ x'\ y'\ z'\ldots$$

After propagating the selectors "infinitely far", we are left with an infinite expression which may involve any functions from $\mathbf{F}^*$, but no others. Our task is to find a finite set of equations with the same solution. We begin by considering the special case in which all functions G in $\mathbf{F}^*$ are *linear*, that is that for any selector S

$$S\ (G\ x\ y\ z\ldots) = (S\hat{}G)\ (S\ x)\ (S\ y)\ (S\ z)\ldots$$

This is not too unreasonable: we can easily show that if G is linear, so are all the S^G. It follows that all functions in $\mathbf{F}^*$ are linear if and only if the original functions $F_1..F_n$ are.

The effect of a selector S on an expression involving only linear functions is particularly simple: every function G is replaced by S^G. That is, the effect of S on such an expression is completely determined by S^. But we showed above that there are only finitely many different S^. Since the a_i can be expressed as such an expression it follows that there are only finitely many different $(S\ a_i)$. Also, if S^ = T^ then we know that $(S\ a_i) = (T\ a_i)$. We can therefore introduce a new variable for each of the $(S\ a_i)$ we need, and express the solution in terms of these variables. We can derive an equation defining $(S\ a_i)$ by pushing S through the definition of a_i. For example, suppose we are solving a single equation

$$a = G\ (CONS_1\ a)$$

and we discover that $\mathbf{F}^* = \{G,\ H\}$ with $CONS_1\hat{} = \{G \mapsto H,\ H \mapsto H\}$. We see that

$$CONS_1\hat{} \circ CONS_1\hat{} = CONS_1\hat{}$$

and so it follows that $(CONS_1\ (CONS_1\ a)) = (CONS_1\ a)$. We introduce one new variable $b=CONS_1\ a$, so $CONS_1\ b=b$, and derive a definition for b:

$$\begin{aligned} b &= CONS_1\ a = CONS_1\ (G\ b) = H\ (CONS_1\ b) \\ &= H\ b \end{aligned}$$

The results of selector elimination read

$$\begin{aligned} a &= G\ b \\ b &= H\ b \end{aligned}$$

In the case where all the F_i are linear, then, we can find an exact solution.

In the non-linear case we cannot find an exact solution, but we can use a similar method to help find an approximate solution that does not overestimate too grossly. We do this by introducing new variables to represent the different $(S\ a_i)$ exactly as in the linear case, but *merging* S a and T (S a) when $T\hat{} \circ S\hat{} = S\hat{}$. This leads to a finite set of selector-free equations as before. Thus we use knowledge about $S\hat{}$ to suggest terms that are "nearly equal", and can probably be merged without losing too much information. In practice most functions in **F*** seem to be linear in most of their parameters, and this approximation seems to give fairly good results.

Eliminating Arrows

Once selectors are eliminated the lion's share of the work is done. The next step is to eliminate arrows, which we do via an abstract interpretation of contexts as subsets of {ABSENT} ∪constructors. The idea is that the abstract interpretation tells us whether each context is (ABSENT ∐ something) or not, and its top-level constructors. The abstract interpretation (#) is defined by

$$\begin{aligned} \bot\# &= \{\} \\ ABSENT\# &= \{ABSENT\} \\ (CON\ c_1..c_n)\# &= \{CON\} \\ (a \amalg b)\# &= a\# \cup b\# \end{aligned}$$

In order to compute the abstract values of all expressions we need abstract versions of ⊓ and →, which we can define by

$$\begin{aligned} (a \sqcap b)\# \quad &= a\# \cap b\# \quad \text{if neither } a\# \text{ nor } b\# \text{ contains ABSENT} \\ &= b\# \quad \text{if only } a\# \text{ contains ABSENT} \\ &= a\# \quad \text{if only } b\# \text{ contains ABSENT} \\ &= a\# \cup b\# \quad \text{if both contain ABSENT} \end{aligned}$$

$$
\begin{array}{lll}
(a \to b)\# & = \{\} & \text{if } a\# = \{\} \\
 & = \{\text{ABSENT}\} & \text{if } a\# = \{\text{ABSENT}\} \\
 & = b & \text{if } a\# \text{ contains a constructor,} \\
 & & \text{but not ABSENT} \\
 & = \{\text{ABSENT}\} \cup b & \text{if } a\# \text{ contains a constructor and ABSENT}
\end{array}
$$

The abstract values of each variable can now be calculated by an iteration, starting with all variables set to $\{\}$, and repeatedly calculating new abstract values for each variable until there is no change. These values can then be used to remove arrows, using the rules:

$$
\begin{array}{lll}
a \to b = \bot & & \text{if } a\# = \{\} \\
= \text{ABSENT} & & \text{if } a\# = \{\text{ABSENT}\} \\
= b & & \text{if } a\# \text{ contains only constructors} \\
= \text{ABSENT} \sqcup b & & \text{if } a\# \text{ contains constructors and ABSENT}
\end{array}
$$

Eliminating $\sqcap$ and $\sqcup$

Before eliminating $\sqcap$s and as many $\sqcup$s as possible it is convenient to introduce new variables for the components of all constructions. This results in equations whose right hand sides are built using $\sqcap$ and $\sqcup$ from variables, ABSENT, and constructions whose components are variables.

Now, as a first stage, we eliminate all top-level variables, so that each right-hand side is formed using $\sqcap$ and $\sqcup$ from ABSENT and constructions only. This is done by choosing any equation and expressing it in the form

$$a = E \sqcup (F \sqcap a)$$

where E and F do not contain a. We can find the least solution of this equation (regardless of the values of the other variables) by iteration. The first approximation is of course $\bot$, and the second is E since $\bot$ is a zero for $\sqcap$ and an identity for $\sqcup$. The third approximation is

$$E \sqcup (F \sqcap E)$$

and we can show that this is always the limit, because the fourth approximation is

$$
\begin{array}{lll}
E \sqcup (F \sqcap (E \sqcup (F \sqcap E))) & & \\
\quad = E \sqcup (F \sqcap E) \sqcup (F \sqcap F \sqcap E) & & \\
 & & (\sqcap \text{ distributes over } \sqcup) \\
\quad = E \sqcup (F \sqcap E) \sqcup (F \sqcap E) & & \\
 & & (\sqcap \text{ is idempotent}) \\
\quad = E \sqcup (F \sqcap E) & & \\
 & & (\sqcup \text{ is idempotent})
\end{array}
$$

We can use this solution to eliminate top level occurrences of a from the other equations.

Repeating the process for each equation, we can eliminate all top-level variables.

As a second stage, we take each equation in turn and push the $\sqcap$ and $\sqcup$ operations inside the constructors using the laws:

$$\begin{aligned}
&\mathrm{CON}\ a_1..a_n \sqcap \mathrm{CON}\ b_1..b_n = \mathrm{CON}\ (a_1 \sqcap b_1)..(a_n \sqcap b_n)\\
&\mathrm{CON}\ a_1..a_n \sqcap \mathrm{CON'}\ b_1..b_m = \bot \qquad\qquad \text{if CON}\neq\text{CON'}\\
&\mathrm{CON}\ a_1..a_n \sqcup \mathrm{CON}\ b_1..b_n = \mathrm{CON}\ (a_1 \sqcup b_1)..(a_n \sqcup b_n)
\end{aligned}$$

This results in a right hand side which is an $\sqcup$ of ABSENT and constructions with distinct constructors. The components of these constructions are "Boolean combinations" of variables. We replace each component by a new "derived" variable, which converts this equation into its final form. We must also add an equation defining each derived variable to the equations waiting to be processed. Each derived variable is defined to be equal to the expression it replaced, with the variables replaced by their values so that the new equation is in the same form as the other unprocessed ones (that is, built from ABSENT and constructors with variables as their components). Since each derived variable is a Boolean combination of older variables, each can also be expressed as a Boolean combination of *underived* variables. There are only finitely many different Boolean combinations of a finite set of variables, and so only finitely many derived variables need be introduced, and the process terminates if we include a check that prevents us introducing two derived variables to represent the same Boolean combination.

Practical Results

A slightly earlier version of the algorithm described here has been implemented in FranzLisp and used to analyse the strictness of simple list processing functions. It is perhaps surprising that, despite the approximations inherent in the method, quite useful results were obtained. Among the contexts calculated by the strictness analyser were

$$\begin{aligned}
&\mathrm{APPEND}_1\ \mathrm{spine} = \mathrm{spine}\\
&\mathrm{APPEND}_2\ \mathrm{spine} = \mathrm{spine}\\
&\mathrm{APPEND}_1\ (\mathrm{cons\ c\ ABSENT}) = \mathrm{nil} \sqcup \mathrm{cons\ c\ ABSENT}\\
&\mathrm{APPEND}_2\ (\mathrm{cons\ c\ ABSENT}) = \mathrm{ABSENT} \sqcup \mathrm{cons\ c\ ABSENT}\\
&\mathrm{LAST\ c} = \mathrm{cons}\ (\mathrm{ABSENT} \sqcup \mathrm{c})\ \mathrm{g}\\
&\textbf{where}\ \mathrm{g} = \mathrm{nil} \sqcup \mathrm{cons}\ (\mathrm{ABSENT} \sqcup \mathrm{c})\ \mathrm{g}\\
&\mathrm{REVERSE\ spine} = \mathrm{spine}
\end{aligned}$$

Each is the best possible result. These were its successes. It is instructive to consider one of its failures also. We attempted to calculate REVERSE (cons c ABSENT), expecting the result

$$\mathrm{g}\ \textbf{where}\ \mathrm{g} = \mathrm{nil} \sqcup \mathrm{cons}\ (\mathrm{ABSENT} \sqcup \mathrm{c})\ \mathrm{g}$$

Instead the strictness analyser found

$$g \textbf{ where } g = \mathsf{ABSENT} \sqcup \mathsf{nil} \sqcup \mathsf{cons}\ h\ g$$
$$h = c \sqcup \mathsf{ABSENT} \sqcup \mathsf{nil} \sqcup \mathsf{cons}\ h\ g$$

which is correct, of course, in that it is an overestimate of the best answer, but is a very gross overestimate indeed. There are two surprising things about this result. First, the strictness analyser appears to believe that the argument of reverse might be a list of lists (or even more deeply nested structure), because the context in which the elements of the argument appear (h) allows for nils and conses. Second, since g is not a strict context, it has failed to discover that reverse is strict in its argument, even though the very first thing reverse does is to evaluate that argument in a **case** expression! This example shows both the approximations in our algorithm at their worst.

The first problem is due to the approximation in selector elimination. Strangely, in this case $(\mathsf{CONS}_2 \circ \mathsf{CONS}_2)$ and $(\mathsf{CONS}_1 \circ \mathsf{CONS}_2 \circ \mathsf{CONS}_2)$ induce the same functions on $\mathbf{F}^*$, and so their results are merged, even though they have different types. It is this which led to a list context and element context being $\sqcup$ed together in the definition of h. The strictness analyser could avoid the problem by using information provided by the type-checker. Selector elimination would not merge terms of different types. Since there are only a finite number of types in any particular program this would not lead to non-termination.

The second problem is due to the approximation of recursive strictness functions by non-recursive ones. It is a little harder to understand, and can be corrected in two different ways, so we will consider it in some detail. The strictness functions involved in this example are

$$\mathsf{REVERSE}\ a = a \rightarrow \mathsf{nil} \sqcup \mathsf{cons}\ (\mathsf{CONS}_1\ (\mathsf{APPEND}_2\ a))\ (\mathsf{REVERSE}\ (\mathsf{APPEND}_1\ a))$$
$$\mathsf{APPEND}_1\ b = b \rightarrow \mathsf{nil} \sqcup \mathsf{cons}\ (\mathsf{CONS}_1\ b)\ (\mathsf{APPEND}_1\ (\mathsf{CONS}_2\ b))$$

The strictness analyser attempts to expand (REVERSE (cons c ABSENT)) and observes that the recursive call of REVERSE does not have a selector applied to a as its argument, so it does not attempt to expand the recursion. Instead it merges the top-level and recursive arguments to give e, and approximates the call by d.

$$e = \mathsf{cons}\ c\ \mathsf{ABSENT} \sqcup \mathsf{APPEND}_1\ e$$
$$d = e \rightarrow \mathsf{nil} \sqcup \mathsf{cons}\ (\mathsf{CONS}_1\ (\mathsf{APPEND}_2\ e))\ d$$

But now the argument to APPEND_1 cannot be reduced to finite normal form before the call is expanded, and so APPEND_1 e is also approximated, by f where

$$f = g \rightarrow \mathsf{nil} \sqcup \mathsf{cons}\ (\mathsf{CONS}_1\ g)\ f$$
$$g = e \sqcup \mathsf{CONS}_2\ g$$

Now the problem is unavoidable. CONS_2 g$\sqsubseteq$g, but ABSENT$\sqsubseteq\mathsf{CONS}_2$ e$\sqsubseteq\mathsf{CONS}_2$ g. So ABSENT$\sqsubseteq$g, which implies ABSENT$\sqsubseteq$f (because f is g$\rightarrow$..), ABSENT$\sqsubseteq$e (because e is .. $\sqcup$ f), and finally ABSENT$\sqsubseteq$d. d is the result of the analysis and cannot be a strict context. This is a general

problem: whenever a recursive call occurs in a lazy context then our non-recursive approximation approximates the context of *all* calls by a lazy one, which prevents the analyser ever assigning a parameter a strict context. This is a serious limitation indeed.

One possible approach is to consider that the analyser was hasty in deciding to approximate the call of REVERSE non-recursively. In this example a little calculation shows that

$$\begin{aligned}&\mathsf{APPEND}_1\ (\mathsf{cons\ c\ ABSENT}) = \mathsf{nil} \sqcup \mathsf{cons\ c\ ABSENT}\\&\mathsf{APPEND}_1\ (\mathsf{nil} \sqcup \mathsf{cons\ c\ ABSENT}) = \mathsf{nil} \sqcup \mathsf{cons\ c\ ABSENT}\end{aligned}$$

and so an exact expansion is indeed possible. The analyser could always attempt such an expansion, and give up only if it failed to find one after a few levels of recursion.

Alternatively, we could try to improve the non-recursive approximation so that it at least does not make this particular blunder. This can be done by replacing all recursive calls of strictness functions (F E) by the equivalent

$$\mathsf{E} \rightarrow \mathsf{F\ (STRICT\ E)}$$

before making the approximation. STRICT finds the strict part of a context, and satisfies

$$\begin{aligned}&\mathsf{STRICT\ ABSENT} = \bot\\&\mathsf{STRICT\ (CON}\ c_1..c_n) = \mathsf{CON}\ c_1..c_n\\&\mathsf{STRICT\ (a} \sqcup \mathsf{b)} = \mathsf{STRICT\ a} \sqcup \mathsf{STRICT\ b}\end{aligned}$$

This ensures that the approximated context of all calls is a strict one, and so prevents the analyser making such an obvious mistake.

However, we cannot introduce a new operator on contexts without showing how to solve equations involving it. This is not so easy in the case of STRICT, because its properties are not very pleasant. In fact,

$$\begin{aligned}&\mathsf{STRICT\ (a} \rightarrow \mathsf{b)} = \mathsf{STRICT\ a} \rightarrow \mathsf{STRICT\ b}\\&\mathsf{STRICT\ (a} \sqcap \mathsf{b)} = \mathsf{(STRICT\ a} \sqcap \mathsf{b)} \sqcup \mathsf{(STRICT\ b} \sqcap \mathsf{a)}\end{aligned}$$

STRICT is in some ways similar to a selector, and we believe that it can be eliminated during selector elimination. In order to be sure that selector elimination still terminates, we need to convince ourselves that the set $\mathbf{F}^*$ is still finite. We can do so by noting that, for any compound selector S,

$$\mathsf{STRICT\ (S\ (STRICT\ a))} = \mathsf{STRICT\ (S\ a)}$$

which allows us to restrict attention to compositions of selectors and STRICT involving STRICT at most once. $\mathbf{F}^*$ can be calculated by forming the closure of $\{F_1..F_n\}$ under the selectors, adding all functions STRICT^F, and then forming the closure of the resulting set under the selectors.

Conclusion

We have shown that it is possible to compute useful strictness information about functions on non-flat domains in a first-order functional language, but there is much scope for further work. Our study of the operators on and properties of contexts seems a little complicated, and relies in places on informal reasoning. A proper treatment would be welcome. Our approach involves several approximations, and it may well be possible to find better approximations or even exact solutions. It is also a matter of urgency to extend the method to higher-order functions, since much manipulation of data-structures is done by higher-order functions in practical programs.

Can our method be applied in a real compiler? We must conclude that, at least in its present form, it cannot. Any method intended for practical use must be able to analyse one call in a relatively short time. Yet our method analyses a call by calling its strictness function, which in turn may call other strictness functions and so on to unlimited depth. Thus one call may take an arbitrarily long time to analyse, which is quite unacceptable in a real compiler. To alleviate this, the body of each strictness function must be simplified once and for all, so that it can be applied in a reasonably short time. This is essential, even if it involves further approximation.

Thus our work is a demonstration of possibility, and a potential basis for further work that may lead to practical strictness analysis of data-structures in the future.

Acknowledgements

I am grateful to Kent Karlsson and Mary Sheeran for studying my incomprehensible drafts with great care, to Lennart Augustsson, Peter Dybjer, Thomas Johnsson, Simon Peyton-Jones, and Phil Wadler for their helpful comments, to the Programming Methodology Group at Chalmers University for a stimulating environment, and to the UK Science and Engineering Research Council for their financial support in the form of a European Research Fellowship.

Addendum

During the workshop of which this is a Proceedings, Phil Wadler discovered a simple and elegant extension of the Mycroft approach to cope with non-flat domains. His work can easily be combined with Burn, Hankin and Abramsky's extension to higher-order functions. A slight disadvantage is that Wadler's approximation is cruder than ours, and so our method can sometimes discover more detailed information. Wadler will report his result in a forthcoming paper.

References

[1] G. L. Burn, C. Hankin, S. Abramsky, *The Theory and Practice of Strictness Analysis for Higher-Order Functions*, General Electric Company/Imperial College London 1985.

[2] C. D. Clack, S. L. Peyton-Jones, *Generating Parallelism from Strictness Analysis*, Internal note 1679, University College London, February 1985.

[3] J. Fairbairn, *Removing Redundant Laziness from Super-combinators*, in *Proceedings of the Workshop on Implementations of Functional Languages*, Aspenæs, Gøteborg, February 1985.

[4] P. Hudak, J. Young, *A Set-Theoretic Characterisation of Function Strictness in the Lambda Calculus*, in *Proceedings of the Workshop on Implementations of Functional Languages*, Aspenæs, Gøteborg, February 1985.

[5] A. Mycroft, *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*, pp 269-281 in *Proceedings 4th International Symposium on Programming*, Lecture Notes in Computer Science, Vol. 83, Springer-Verlag, Paris, April 1980.

[6] S. C. Wray, *A New Strictness Detection Algorithm*, in *Proceedings of the Workshop on Implementations of Functional Languages*, Aspenæs, Gøteborg, February 1985.