Demand analysis

Simon Peyton Jones, Peter Sestoft, and John Hughes

July 26, 2006

1 Introduction

Any decent optimising compiler for a lazy language like Haskell must include a strictness analyser. The results of this analysis allow the compiler to use call-by-value instead of call-by-need, and that leads to big performance improvements. It turns out that strictness analysis is an interesting problem from a theoretical point of view, and the 1980's saw a huge rash of papers on the subject. There were fewer, many, many fewer, papers that described real implementations.

This paper presents the fruits of a decade-long experience with strictness analysis, in the context of the Glasgow Haskell Compiler, an optimising compiler for Haskell. In particular, we recently re-engineered the existing strictness analyser that used forward abstract interpretation, replacing it with a new one that uses backward analysis instead.

In one sense therefore, this paper contains nothing new: we apply well-understood backward-analysis techniques. However, it turns out that the application is not at all straightforward, and we make the following contributions:

- Beyond strictness analysis, we show that it is essential to perform *absence analysis*. The goal is to pass only the needed parts of a value in a function call, and to perform unboxing, passing only naked machine integers instead of boxed values when possible.
- While we introduce these two analyses separately, we show how to combine them into a single analysis over a richer domain; see §B. Until now, GHC has had to do two separate analyses. (In fact, a third analysis, Constructed Product Result analysis, fits in beautifully as well, so in reality the new analyser does all three analyses at once. CPR analysis is described elsewhere [?], and we do not discuss it further in this paper.)
- Backwards analysis eschews the accuracy that can be achieved by higher-order abstract interpretation, but it is a great deal faster, especially for deeply-nested functions. How much faster? And how much accuracy is lost? We give some indicative answers in §C.
- Although backwards analysis is not higher order, in the sense that it does not track the effect of functional arguments, our analysis must apply to a higher-order language (Haskell). We give a new and elegant formulation of backwards analysis for a higher-order language, based on "*call demands*" in §4.1.4. This formulation leads directly to a rather compact implementation.

  Furthermore, the implementation has proved to be reassuringly generic; during development of the new analysis we repeatedly changed the domain and its two operations ("lub" and "both") while hardly changing the analysis function at all.
- Implementing our new analysis in a real compiler forced us to confront several issues that were completely hidden before we tried the implementation. Two particular examples are: correct analysis of the `error` function (§2.2); and accurate analysis of nested function definitions, which are very common in GHC (§5.3).
- We use a clever folk-lore technique to improve the speed of convergence, and give measurements of its effectiveness (§9).
- We describe and motivate a range of engineering design choices. For example, we have found that it is very important to "look inside" products (§4.1.2); that nested definitions are very common and must be handled well (§5.3); and that simple approximations to the full demand transformer for a function work well in practice (§5.1).

The resulting analyser is half the size of its predecessor, is much easier to understand and modify, and runs much faster.

The focus of the paper is strongly practical. The theory supports a wide spectrum of analyses, ranging from accurate-but-expensive to cheap-but-coarse. The context of a real compiler guides our choices in this multi-dimensional space. We give a formal specification of the domains, their operations, and the analysis function itself, but we do not attempt to prove soundness with respect to a more abstract specification, leaving that for further work.

2 Characterising the problem

The default parameter-passing mechanism in Haskell is call-by-need, in which an argument must be passed as a thunk, or 'box', encapsulating the argument expression. At the first use of the parameter, the thunk is evaluated and overwritten with the result, which is then ready at all later uses.

An optimising Haskell compiler can often replace this general calling mechanism by a specialised, more efficient one:

- *Using call-by-value.* When the called function will definitely evaluate its argument, the caller can evaluate the argument early and pass the value itself instead of a thunk. Program analyses that find such *strictness* information have been studied intensively [?, ?, ?, ?].

- *Unboxing arguments.* If the called function needs only the components of a tuple, not the tuple itself, then the caller can pass the components instead of building a tuple. For example:

  ```
  lenFst :: ([a],b) -> Int
  lenFst x = case x of { (p,q) -> length p }
  ```

  Here, `lenFst`'s caller can not only evaluate the argument, because `lenFst` is strict, but also extract the components of the pair and pass only the first one to `lenFst`. A call site like `(lenFst (g t))` can then be transformed to

  ```
  case (g t) of { (p,q) -> $wlenFst p }
  ```

  where `$wlenFst` is the specialised-calling-convention version of `lenFst`.

  Program analyses that find such so-called *boxing* information or absence information have been described in [?, ?].

The compiler's task splits into two: (a) perform a static *demand analysis* of the program, and (b) exploit the information thus discovered. In the literature, much more attention is paid to (a) than to (b), yet one can only understand what information we need from the demand analyis by understanding the use to which that information is put. So we focus initially on (b), to provide the context for the design decisions we subsequently make for the analysis itself.

2.1 The worker-wrapper split in GHC

In GHC, the results of demand analysis are exploited in two ways:

- It drives the *worker-wrapper transformation*, which exposes specialised calling conventions to the rest of the compiler. In particular, the worker-wrapper transformation implements the unboxing optimisation.

- During code generation, the code generator uses call-by-value for strict functions, instead of call-by-need.

The worker-wrapper transformation splits each function `f` into a *wrapper*, with the ordinary calling convention, and a *worker*, with a specialised calling convention. The wrapper serves as an impedance-matcher to the worker; it simply calls the worker using the specialised calling convention. The transformation can be expressed directly in GHC's intermediate language. Suppose that `f` is defined thus:

```
f :: (Int,Int) -> Int
f p = <rhs>
```

and that we know that `f` is strict in its argument (the pair, that is), and uses its components. What worker-wrapper split shall we make? Here is one possibility:[1]

```
f :: (Int,Int) -> Int
f p = case p of
        (a,b) -> $wf a b

$wf :: Int -> Int -> Int
$wf a b = let p = (a,b) in <rhs>
```

Now the wrapper, `f`, can be inlined at every call site, so that the caller evaluates `p`, passing only the components to the worker `$wf`, thereby implementing the unboxing transformation.

But what if `f` did not use `a`, or `b`? Then it would be silly to pass them to the worker `$wf`. Hence the need for *absence analysis*. Suppose, then, that we know that `b` is not needed. Then we can transform to:

```
f :: (Int,Int) -> Int
f p = case p of (a,b) -> $wf a

$wf :: Int -> Int
$wf a = let p = (a,error "abs") in <rhs>
```

Since `b` is not needed, we can avoid passing it from the wrapper to the worker; while in the worker, we can use `error "abs"` instead of `b`.

There's a more obvious problem, though: we seem to take apart `p` in the wrapper, only to rebuild it in the worker. We describe the re-construction of `p` in the worker as *reboxing*; it is plainly a Bad Thing.

However, *the idea is that since* `<rhs>` *is strict in* `p` *it must presumably be taking it apart.* So inside `<rhs>` we may see "`case p of ...`". Since `p` is explicitly bound to a pair in `$wf`, we can eliminate the `case` in `<rhs>`, and that in turn will usually mean that `p` is dead, and the reboxing can be discarded. For example, suppose `f` was like this:

```
f :: (Int,Int) -> Int
f p = (case p of (a,b) -> a) + 1
```

Then the worker-wrapper transformation will produce:

```
f :: (Int,Int) -> Int
f p = case p of (a,b) -> $wf a

$wf :: Int -> Int
$wf a = let p = (a,error "Urk")
        in (case p of (a,b) -> a) + 1
```

Now, in the code for `$wf`, we can inline the definition of `p` at its use in the `case`, simplify the case, and discard the now-dead binding for `p`, giving:

```
$wf :: Int -> Int
$wf a = a + 1
```

Does the reboxing binding still disappear if `p` is not scrutinised by an explicit `case`? For example, what if it is instead passed to another strict function, `g`? In that case `g` will get a wrapper that takes the pair apart; that wrapper will get inlined into `$wf`, and the `case` will cancel as before. Is *all* reboxing eliminated in this way? No, it is not, a problem that we discuss in §2.3.

In short, the worker-wrapper transformation allows the knowledge gained from strictness and absence analysis to be exposed to the rest of the compiler simply by performing a local transformation on the function definition. Then

[1] A real compiler would avoid splitting very small functions, such as `f` above, since they can be inlined bodily, which is better than splitting. For presentational purposes we use small examples regardless of this; you can always make them bigger!

ordinary inlining and case elimination will do the rest, transformations the compiler does anyway. More details are in [?, ?].

2.2 `seq` and `error`

Demand analysis in Haskell is made trickier by two functions that are part of Haskell 98: `error` and `seq`. We briefly introduce their difficulties here, by way of background.

The Haskell 98 function `error :: String -> a` takes a `String`, prints the string, and brings execution to a halt[2]. From a semantic point of view, `error s` should be considered identical to $\bot$, or divergence. For example, consider this function:

```
f []     y = error "urk"
f (x:xs) y = y
```

Is it safe to use call-by-value for `y`? Yes, because `f` either evaluates `y` or else calls `error "urk"`. If we use call-by-value, the call `(f loop)`, where `loop` goes into a loop, will diverge instead of printing "`urk`", but we deem that acceptable behaviour; the program goes wrong in either case, and we allow the compiler to change the particular manifestation of going-wrong-ness.

However, consider these two functions:

```
g1 x y = g1 y x
g2 x y = error x
```

The first function goes into a loop, and does not use either of its two arguments. We could safely treat them as absent, and not pass them at all. The second function also "diverges"; it does not use `y`, *but it does use* `x`. Even though `error` "diverges", you must pass its argument so that it can be printed. More concretely, it is not acceptable to perform a worker/wrapper split for `g2` like this, because although both produce an error, the produce different messages:

```
g2 x y = $wg2
$wg2 = let x = error "abs"
           y = error "abs"
       in error x
```

In short, we must be careful not to assume that `x` is absent simply because it is consumed by a "divergent" computation.

A different difficulty is raised by `seq :: a -> b -> b`, which evaluates its first argument before returning its second. The existence of `seq`, with a polymorphic type, has a pervasive effect. For example, eta reduction is not valid in general:

$$\texttt{g1 a b} \neq \texttt{(\x -> g1 a b x)}$$

The former is $\bot$, while the latter is not, and the two can be distinguished by `seq`. *SLPJ: Need to explain why this makes things difficult. John: Does it make things difficult? Isn't the only implication that, for function demands, $S(S) \neq S(L)$? It would be more of a problem if they were the same!*

[2] In GHC, `error` raises an exception, a nice generalisation of the Haskell 98 behaviour [?].

2.3 Shortcomings of the existing analyser

For some years, GHC has used an analyser based on the classic technique of abstract interpretation [?, ?] to derive strictness and absence information; this information in turn drives the generation of specialised calling conventions. The existing analyser is described in our earlier papers [?, ?].

The worker-wrapper transformation works fine, but the preceding analysis phase, which drives the worker-wrapper transform, is very slow for deeply nested definitions. Given:

```
f x y z = <rhs>
```

the analyser figures out whether `f` is strict in `x`, `y`, and `z` by computing (`f` $\bot$ $\top$ $\top$), (`f` $\top$ $\bot$ $\top$), and (`f` $\top$ $\top$ $\bot$), where $\top$ is the top-most abstract value, and $\bot$ is the bottom-most. If `f` is recursive, it iterates the process using the newly-computed approximation to `f`. The difficulty here comes when `<rhs>` contains nested recursive definitions. Then to compute (`f` $\bot$ $\top$ $\top$), for example, we must compute the abstract values of the nested definitions, given these particular bindings: `x` $= \bot$, `y` $= \top$, and `z` $= \top$. And then do it all again for the next set of bindings. Computing these abstract values itself involves the same sort of iterative process for each recursive nested definition. Result: the running time is exponential in the nesting depth of definitions. This problem can be fixed, but that would further complicate the analyser. Backwards analysis is, as we shall see, much more efficient.

Furthermore, once we looked into it, we found that we could express the backwards analysis rather elegantly. As a direct result, the new analyser is significantly shorter than its predecessor (in source code terms). Even if it were no more efficient, this would be a worthwhile gain. (This is, of course, a "soft" claim: perhaps a re-engineered version of the forwards analysis would be equally concise.)

3 Evaluation demand and usage demand

The preceding section should have convinced you that we want two sorts of information from our demand analysis:

- *Evaluation demand*, or *strictness*, describes the extent to which the expression is guaranteed to be evaluated. The compiler uses strictness information to replace call-by-need with call-by-value.
- *Usage demand*, or *absence*, describes what parts of the expression's value are used. The compiler uses usage-demand information to decide which fragments of the argument to pass to the specialised version of the function.

Absence analysis would, for example, distinguish `g1` and `g2` in the preceding section.

Strictness analysis is well understood, so we tackle that first, in §4. Doing so gives us a chance to discuss several important design choices in a familiar framework. Then, in §7, we describe our absence analysis; the analysis is less familiar, but the framework is identical to that for strictness analysis. Finally, in §B we show that the two can be combined into a single analysis that does the whole job in one blow.

4 Strictness demands

The basic purpose of the strictness analyser is to find the *strictness* that a function places on each of its arguments. We summarise these demands in the function's *demand signature*. For example:

```
null :: [a] -> Bool
-- Demand sig: S
null v = case v of
           []     -> True
           (x:xs) -> False

swap :: (a,b) -> (b,a)
-- Demand sig: S(L,L)
swap p = case p of
           (x,y) -> (y,x)

fst :: (a,b) -> a
-- Demand sig: S(S,L)
fst p = case p of
          (x,y) -> x

f :: Int -> Int -> Int
-- Demand sig: SL
f x y = x+1
```

The demand signatures — so far describing only strictness — are given in comments. Informally, demand `S` (strict) means that the function definitely evaluates the argument; `L` (lazy) means that it may or may not evaluate the argument; and e.g. $\mathtt{S}(s_1, s_2)$ means that it definitely evaluates the argument pair, and evaluates its components to a degree described by s_1 and s_2.

In the rest of this section we establish precisely what we mean by a *demand*, returning to *demand signatures* in Section 5.

4.1 The strictness domain and its operations

A demand analyser answers the question "what demand does the function place on its argument?". A simple demand analyser can work with a simple lattice of demands, such as "definitely evaluated" (strict) and "possibly not evaluated" (lazy). When the type of the argument is known, a more sophisticated analyser may want to represent richer demands. For example, for pairs we may want to be able to say "strict in the pair, strict in the first component, but lazy in the second component".

So a key aspect of the demand analyser is the design of the domain of demands. In this section we describe and motivate this design.

The syntax of demands is shown in Figure 1. It is not a free algebra; the same Figure gives identities that we shall use freely. The demands have the following intuitive meanings:

L is a *lazy* demand. If an expression e places demand L on a variable x, we can deduce nothing about how e uses x. L is the completely uninformative demand, the top element of the lattice.

S is a *head-strict* demand. If e places demand S on x then e evaluates x to at least head-normal form; that is, to the outermost constructor of x. The demand $S(L \ldots L)$ places a lazy demand on all the components, and so is equivalent to S; hence the identity $S = S(L \ldots L)$.

$S(\overline{s})$ is a structured demand on a product (§4.1.2) or function (§4.1.4). It is at least head-strict, and perhaps more.

$\bot$ is a *hyperstrict* demand. The expression e places demand $\bot$ on x if every evaluation of e is guaranteed to diverge, regardless of the value of x. We call this demand "hyperstrict" because it is safe to evaluate x to arbitrary depth before evaluating e.

A demand that is hyperstrict on any component of a

The set *StrDmd* is the set of strictness, or evaluation, demands s defined thus:

s	::=	$\bot$	Hyperstrict
	|	L	Lazy
	|	S	Strict
	|	$S(\overline{s})$	Product or function is evaluated, components $\overline{s}$
$\overline{s}$	::=	$[s_1, \ldots, s_n]$	Tuple components

Identities on demands

$$S(L \ldots L) = S \qquad S(\ldots \bot \ldots) = \bot$$

Ordering on demands

$$\bot \sqsubseteq s \qquad S(\overline{s_1}) \sqsubseteq S(\overline{s_2}) \text{ if } \overline{s_1} \sqsubseteq \overline{s_2} \qquad s \sqsubseteq L$$

The evaluation demand $s_1 \sqcup s_2$ means 'either demand s_1 or demand s_2'. It combines demands from two 'alternative' sub-expressions, exactly one of which is demanded, such as the branches of a `case`. It is the 'least upper bound' operator for the $\sqsubseteq$ ordering.

$$\begin{array}{rcll} \bot & \sqcup & s & = s \\ L & \sqcup & s & = L \\ S(\overline{s_1}) & \sqcup & S(\overline{s_2}) & = S(\overline{s_1} \sqcup \overline{s_2}) \end{array}$$

The evaluation demand $s_1 \& s_2$ means 'both demand s_1 and demand s_2'. It combines demands from two 'parallel' sub-expressions, both of which are demanded.

$$\begin{array}{rcll} \bot & \& & s & = \bot \\ L & \& & s & = s \\ S(\overline{s_1}) & \& & S(\overline{s_2}) & = S(\overline{s_1} \& \overline{s_2}) \end{array}$$

Figure 1: Evaluation demands

Figure 3: Strictness for lifted integers `Int`

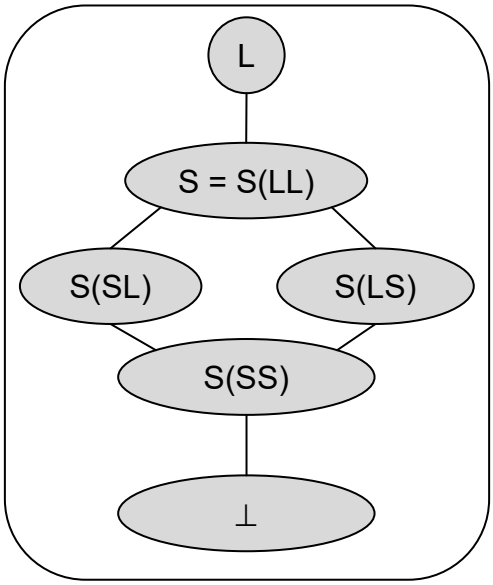

Figure 4: Strictness for (integer) pairs

machine integers. So the strictness lattice for (lifted) integers `Int` can be computed from Figure 2, together with the identities of Figure 1, thus:

$$\begin{array}{rcl} \mathcal{T}_s[\![\texttt{Int}]\!] & = & S(\mathcal{T}_s[\![\texttt{Int\#}]\!]) \cup \{L\} \\ & = & \{S(\bot), S(L), L\} \\ & = & \{\bot, S, L\} \end{array}$$

The resulting lattice is shown in Figure 3.

Using the same construction again, we can derive the lattice for the type `(Int,Int)`, shown in Figure 4. In constructing this Figure, we again make use of of the identities of Figure 1.

4.1.3 Sum types

In contrast to products, the worker-wrapper transform cannot take advantage of information about the components of a sum type (an algebraic data type with more than one constructor). For example:

```
f :: Maybe Int -> Int
f (Just n) = n
f Nothing  = 0
```

Clearly, `f` is strict, and the code generator can take advantage of that when compiling calls, but it is hard to do any unboxing. To do so would mean passing *either* `n` *or* nothing at all, plus a tag to say which was the case. This is indeed possible, but it is tricky — for example, it is harder to tell the garbage collector where the pointers are, since it may depend on the value of the tag — and it is hard to express in GHC's intermediate language. For a sum type, we therefore use a simple three-point lattice $\{\bot, S, L\}$. In the Figure, we take the list type as an example.

Aside. Even in the absence of unboxing, though, one could argue that the analyser could usefully compute structured strictness information. For example, if `f` above is given a `Just` argument, then it is certainly strict in the argument to the `Just`. So if we saw the call `(f (Just (h x)))` we could compute `(h x)` by value, before boxing it in a `Just`. A great deal of work has been done on analyses (especially backwards analyses) that can "look inside" sum types, including recursive data types [?]. However, our initial focus was on the low-hanging fruit, so we decided to leave sum types for further work. *End of aside.*

4.1.4 Function types and call demands

Consider the curried application `(f x y)`, which really means `((f x) y)`. If demand d is placed on this expression, what demand is placed on the sub-expression `(f x)`? One possibility would be to finesse the question by regarding the expression as the application of `f` to two arguments, rather than treating it as a curried application, but it turns out that the analysis works very elegantly if we take the curried approach and answer the question directly.

Clearly, if `((f x) y)` is evaluated with strict demand d, then `(f x)` is evaluated with demand "evaluate `(f x)` to a function, call it, and place demand d on its result". We denote this demand on `(f x)` by the *call demand* $S(d)$. So the demand on `f` itself is $S(S(d))$. (This is rather natural, if we think of a function as a (possibly infinite) tuple of all its possible results). We believe that this use of call demands is new to this work – but see Section E. We use the same notation, $S(d)$, for functions as for products, relying on the type-indexing of the domain to ensure that a call demand never "meets" a product demand.

These call-demands can show up in demand signatures. For example, consider:

```
app :: Int -> (Int->Int) -> Int
app x f = f x
```

We can see that `app` will place a call demand on `f`, and evaluate the result, so `app`'s demand signature is `LS(S)`.

One could imagine being cleverer: if `f` is strict, then `app` is strict in its first argument also. Indeed, higher-order abstract interpretations discover exactly such information [?]. However, despite considerable attention in the 1980's, we believe that it is a poor choice to seek this higher-order information.

Firstly, higher-order analyses are expensive: the analysis is sophisticated and the function-space domains are large, which in turn leads to slow fixpoint calculations. Second, it is awkward to convey the inter-argument dependence across separate compilation boundaries. Third, the worker-wrapper split would not be aided by such information, since the splitting is driven solely by the function's *definition*, not its call sites. Fourth, in our experience, in many cases where it matters, the higher order function is itself small enough to inline before the demand analyser runs, so the effort is in vain.

All this is fortunate, because inter-argument dependence is in any case inaccessible to a purely backwards analysis. Our conclusion is that this apparent shortcoming of backwards analysis is hardly a drawback in practice.

4.1.5 Polymorphic types

Haskell is a polymorphic language, so the strictness analyser will not always know the type of the expression it is analysing. For example, consider the function `k`:

```
k :: a -> b -> a
k x y = x
```

Clearly, `k` is strict in its first argument, and lazy in its second, so it has the evaluation-demand signature `SL`. Another very important polymorphic function is `seq :: a->b->b`, whose demand signature is `SS`; it is strict in both its parameters.

So the appropriate demand lattice for an unknown type, represented at compile type by a type variable α, is the same as that for sum types, namely $\{\bot, S, L\}$. Notice that this lattice is a sub-lattice of every other demand lattice, except that for unlifted types. This is as we expect – we simply approximate more vigorously for polymorphic types. Moreover, when we "instantiate" a polymorphic demand at a particular type, there is no work to do, since the polymorphic demand is already a valid demand for the instance type. (Fortunately, GHC does not allow polymorphic type variables to be instantiated to an unlifted type [?], since that would preclude compiling polymorphic functions to just one generic piece of code).

4.2 Operations on demands

This completes our tour of Figure 2. Notice that in all cases, L is the top element of the demand lattice; it is always a safe approximation, and means "no information".

The ordering among demands is given in Figure 1. As usual, $s_1 \sqsubseteq s_2$ means that s_1 denotes at least as strong a demand as s_2. The demands relevant for a particular expression depends on the type of that expression, as we have already seen.

There are two key operations over demands, also given in Figure 1:

- $d_1 \sqcup d_2$ combines two *alternative* demands for a value, such as the demands arising from branches of an `if` or `case`. For example:

```
case x of
  []     -> y
  (p:ps) -> True
```

If this expression is evaluated with demand S, the first branch places demand S on `y`, while the second branch places demand L (lazy or absent); the overall expression therefore places demand $S \sqcup L = L$ on `y`.

Of course, the $\sqcup$ operation can be derived directly from the ordering relation. We write it in Figure 1 here only for completeness.

- $d_1 \,\&\, d_2$ combines two demands, *both* of which are placed on the value. For example, consider the expression

```
f x x
```

where `f` has demand signature `LS`. If the expression is evaluated with demand S, then `x` will be consumed by both demand L and demand S. The aggregate demand on `x` is therefore $L \& S = S$.

For another example, consider:

```
fst x + snd x
```

The first subexpression places a strict demand `S` on the first component of `x`, and therefore places demand `S(SL)` on `x`. Similarly, the second subexpression places demand `S(LS)` on `x`. The total demand on `x` therefore is `S(SS)`.

We may need to use the demand identities of Figure 1 to perform these computations. For example:

```
seq x 3 + fst x
```

Computing demands on `x` will give

$$\begin{aligned} S \,\&\, S(SL) &= S(LL) \,\&\, S(SL) \\ &= S(SL) \end{aligned}$$

The polymorphic demand S from the polymorpic `seq` meets the demand $S(SL)$ from the `fst x`. First, use the identity $S = S(LL)$ to expand the S demand, and then & can be done elementwise.

Both $\sqcup$ and & are commutative and associative, and enjoy the following distributive property:

$$s_1 \,\&\, (s_2 \sqcup s_3) = (s_1 \,\&\, s_2) \sqcup (s_1 \,\&\, s_3)$$

In the case of strictness analyis, the & operator is greatest lower bound (glb), but that will not always be the case, which is why we give it a different name.

5 Demand signatures

Demand signatures play a central role in our analysis. The demand signature for a function summarises all the information that the analyser computes about that function. In particular:

- The demand signature of a function is all the analyser knows about a function when it encounters a call site.
- The demand signature is the information that is exported across separate compilation boundaries.
- The demand signature embodies all the information necessary to make the worker-wrapper split for a function.
- The demand signature tells the code generator when it can use call-by-value at a call site.

So far we have only given an informal intuition for the meaning of a demand signature. This section makes it precise.

5.1 Demand signatures as demand transformers

We begin with the following key idea, which explains what a demand signature means: *a demand signature for a function is simply a compact approximation to the function's demand transformer.*

Backwards analysis of a function aims to answer the following question: "given demand d on the function's result, what are the demands on the function's argument(s)?". The stronger the demand on the function, the stronger the demand on the arguments. For this purpose, a function may be seen as a monotonic *demand transformer*: it transforms a demand on the function's result into demands on the function's argument (and free variables, see §5.3). Consider this example:

```
f :: [a] -> [a] -> Bool
f xs ys = null xs && null ys
```

Informally, we would say that `f` is strict in `xs` and has demand signature `SL`. But consider the expression

```
f (error "urk") `seq` True
```

Even though `f` is strict in its first argument, evaluating the partial application `f a` does not force evaluation of `(error "urk")`. Only when `f` has been given *both* its arguments does it unleash the strict demand on its first argument.

The optimal results produced by the demand transformer for `f` are shown in the following table:

	Demand on	
Function	First arg	Second arg
`S`		
`S(S)`	`L`	
`S(S(S))`	`S`	`L`

If the function `f` itself is simply evaluated (presumably by `seq`), then it imposes no demand on its arguments. If it is applied to a single argument, only a lazy demand is placed on that argument. If it is applied to both arguments, then a strict demand is placed on the first one, and a lazy demand on the second one.

Here is a more elaborate example:

```
g :: (Int,Int,Int) -> [a] -> (Int,Bool)
g (a,b,c) = if a==0 then error "urk"
            else \y -> if b then (c, null y)
                            else (c, False)
```

Function `g` consumes a single argument (a pair), pattern-matches on it, and evaluates its first component `a`. Then `g` consumes its second argument, and tests `b` before returning a pair. If that pair is itself evaluated, then `c` and/or `ys` will

be evaluated. The following table shows part of an optimal demand transformer for `g`:

	Demand on	
Function	First arg	Second arg
`S`		
`S(S)`	`S(SLL)`	
`S(S(S))`	`S(SSL)`	`L`
`S(S(S(SL)))`	`S(SSS)`	`L`

A sophisticated backwards analyser could capture the full glory of these demand transformers, but our analyser instead uses a brutal, yet effective, approximation:

- Find the syntactic arity, n, of the function; that is, how many explicit lambdas it has at the top of its right-hand side. For example, `f` above has syntactic arity 2, while `g` has syntactic arity 1.
- Compute the demand placed on the arguments by the *vanilla call demand* $S(\ldots S(S)\ldots)$, where the call demands are nested n deep.
- Record the demands thus computed as the function's demand signature. For example, `f`'s demand signature is `SL` and `g`'s is `S(SLL)`. Thus `f`'s demand signature represents the demand transformer shown in the table pretty well. However, `g`'s demand signature loses all information about how demand on the result of the first application propagates to demand on the `b` and `c` components of the argument. The last two lines of the table for `g` could not be derived from `g`'s demand signature.
- At a call site, use the demand signature as an emasculated demand transformer in the following way. If the demand on the function at a call site is weaker than the vanilla call demand, place demand `L` on all arguments. Otherwise place the demands specified by the demand signature on the arguments.

In effect, we represent the entire demand transformer function by a single (argument, result) pair. This approach is sound (because it can only under-estimate demands), but it is clearly approximate, as shown by the `g` example. Nevertheless, it is a fine thing to have a compact representation for a function's demand transformer. First, it makes fixpointing faster, because the domain is less rich; and second, it makes it easy to export information across module boundaries.

We believe that this approximation works extremely well in practice. This belief is based on eye-balling the output of the demand analyser for many programs. The only way to be sure that the benefit of a more sophisticated representation would be slight is to try it — and we have not done that.

5.2 Demand signatures: the full story

So far, we have informally implied that a demand signaure is just a sequence of demands. For example, we have written signatures such as

```
f :: Int -> Int -> Int
-- Demand sig: SL
f x y = x+1
```

It is already clear that a demand signature must also include some measure of the depth of the vanilla call demand discussed in the previous section. So `f`'s demand signature might more properly be described thus:

$$\langle 2, \texttt{SL} \rangle$$

This is not enough, however. In the following subsections we will show that demand signatures must be elaborated in two distinct ways. First, we must add information about free variables (§5.3); and second, we must add information about divergence (§5.4).

5.3 Nested definitions and thunks

Our demand analyser may encounter nested function definitions, such as this one:

```
f b x y = let
            g z = x + y + z
          in
          if b then y else g (x*x)
```

Here `g` is defined locally, inside `f`'s right hand side. Nested function definitions like this one are common in user-written code, and even more common once the compiler has done some inlining and let-floating.

At first one might think that dealing with nested definitions is easy: simply compute `g`'s demand signature, and then deal with the body of the `let`. But the only occurrence of `x` is in the right hand side of `g`, so we must somehow take into account the *free variables* of `g`.

How might we compute the demand on `x`? One way is as follows: treat a `let` as syntactic sugar for a lambda. Desugaring the `let` would give:

```
f b x y = (\g -> if b then y else g (x*x))
          (\z -> x + y + z)
```

The first lambda is lazy in `g` (since the conditional is not sure to call `g`, so the analyser will analyse the argument `(\z -> x + y + z)` with a lazy demand, and hence derive a lazy demand on both `x` and `y`.

Unfortunately, this answer is over-pessimistic for `y`. Either `b` is `True`, in which case `y` is evaluated, or `b` is `False`, in which case `g` is called, and `y` is evaluated. So `f` is certainly strict in `y`. Even worse, the `let`-as-lambda approach does not even expose the obvious fact that `g` is strict in its first argument, so the call `g (x+x)` can use call-by-value.

In short, treating `let` as if it were a lambda gives sound results, but it must surely be better to treat `let` directly. At least the direct approach can analyse the right-hand-side of `g` and extend the environment with `g`'s demand signature before analysing the body; that will expose the fact that `g (x*x)` can use call-by-value. To capture the strictness in `y` as well, we compute a richer demand signature for `g`, one that embodies not only the demand it unleashes on its argument, `z`, *but also the demand it unleashes on its free variables.* We write this richer signature thus:

$$\langle 1, [\texttt{x} \mapsto S, \texttt{y} \mapsto S] \Rightarrow [S] \rangle$$

This says that if `g` is applied to one argument (the "1" in the signature), it unleashes demand S on that argument (the

part after the "⇒"), *and also demand S on* `x` *and* `y` (the finite mapping "[...]").

One possible alternative to these complications is to finesse them by performing lambda lifting, so that all function definitions are at top level. Then our example would become:

```
g' x y z = x + y + z
f b x y = if b then y else g' x y (x*x)
```

Now it is clear that `y` is used in both branches of the conditional, while `x` is used in only one. Indeed, we can see the richer demand signatures as a simulation of the extra parameters introduced by lambda lifting.

The trouble is that this approach does not work for *thunks*. Here is another version of the same example:

```
f b x y = let
            z = x + y
          in
          if b then y else z
```

Unlike `g`, `z` has no parameters, and in general we cannot lambda-lift such definitions without losing sharing. Yet, `f` is still strict in `y` and lazy in `x`. To compute this strictness, we require the richer demand signatures for thunks, embodying demands on free variables.

Earlier works on backwards analysis assumed lambda-lifting for functions, but how did they deal with thunks? Answer: using the let-as-lambda approach, thus:

```
f b x y = (\z -> if b then y else z) (x+y)
```

That is, first compute the strictness of `(\z -> ...)`, and then apply that to `(x+y)`. In this case the abstraction is lazy, so we will (safely but imprecisely) conclude that `f` is lazy in `y`.

We believe that our approach is quite new. By enriching demand signatures with free-variable demands we are able

- To treat thunks uniformly. Indeed, our implementation has little notion of a "local function definition". Rather, it deals separately with local definitions (`let` and `letrec`) on the one hand, and lambda abstractions on the other.
- To obviate the need for lambda lifting. Lambda lifting is undesirable, because it moves the nested function's code out of its context, sometimes losing optimisation opportunities.
- To get more accurate results, as we have demonstrated in the examples above. We have found this additional accuracy to be quite significant in practice. *SLPJ: Todo: quantify in the experience/implementation section.*

In essence, we have added just a little bit of forwards analysis to our otherwise backwards analyser, and found that this little bit is both cheap and effective.

5.4 Divergence and the `error` function

As well as knowing what a function does with its argument(s) it is also essential to know something about its result. Consider the evaluation demand placed on `y` by this function:

```
f True  y = y
f False y = error "urk"
```

Would it be safe to use call-by-value for both arguments to `f`? Yes, because `error` always diverges, so evaluating `y` early is safe regardless which of the branches is taken: if `y` does not terminate, then `f` would not terminate anyway. So, even though `y` is not mentioned in the right hand side `error "urk"`, we must consider that `error "urk"` places a strict demand on `y`.

It would be possible to make `error` a special case in the analysis, but programs often feature "dressed up" versions of `error`:

```
myError s = error ("Fatal error: " ++ s)
```

Rather than treating `error` as a special case, we instead embody the 'always-diverges' information in the demand signature of the function, thus:

$$\texttt{error} : \langle 1, \bot \Rightarrow [S] \rangle$$

Here, we have added a new component to the demand signature, a single demand written before the "⇒", that we call the *result demand*. At a call site, the result demand r is unleashed on all the variables in scope at the call site.

A result demand can only take two values:

Unknown: L. We can assume nothing about the result.

Diverges: $\bot$. The function is guaranteed to diverge, or to raise an error.

John: Interesting semantics: projects not only the environment at the definition, but the environment at the call! Given that `(error x)` diverges, however, one might wonder whether it makes any difference whether `error x` is considered to be strict in `x` or lazy in `x`. More concretely, is there any difference between these two demand signatures?

$$\langle 1, \bot \Rightarrow [S] \rangle$$
$$\langle 1, \bot \Rightarrow [L] \rangle$$

No, there is not: since the result demand is $\bot$, the argument demand is irrelevant. *SLPJ: Do we need to say more about this? Result demand is placed on all in-scope variables. Fwd reference to absence. Give signature equivalences.*

5.5 Summary

Motivated by the preceding sections, Figure 5 gives the syntax for demand signatures: a demand signature is simply a pair of an arity, n, and a *demand type*.

A *demand type* encodes the demands unleashed by a function on its context when it is applied to enough arguments. As Figure 5 shows, a demand type has the form $\theta \Rightarrow [s_1 \ldots s_n]$, consisting of two components:

- A *demand environment* θ, that gives the demand placed by e on the variables in scope at the call site.
- A sequence of demands, $[s_1 \ldots s_n]$, which give the demands that e places on its arguments. The sequence is non-empty only for function-typed expressions.

Demand signatures

$$sig \quad ::= \quad \langle n,\ dt \rangle$$

Demand types

$$dt \quad ::= \quad \theta \Rightarrow \sigma$$
$$\sigma \quad ::= \quad [s_1, \ldots s_n]$$

Particular demand types

$$AbsType \quad = \quad \langle [\,], ID_{\&} \rangle \Rightarrow [\,]$$
$$BotType \quad = \quad \langle [\,], \bot \rangle \Rightarrow [\,]$$

Demand type equivalence

$$\theta \Rightarrow [\,] \quad \equiv \quad \theta \Rightarrow [\top]$$

$$[s_1, \ldots, s_n] \ \sqcup \ [t_1, \ldots, t_m]$$
$$= \quad [s_1 \sqcup t_1, \ldots, s_k \sqcup t_k]$$
where $k = min(n, m)$

A result demand, r, is either $\bot$ (the identity of $\sqcup$), or $ID_{\&}$ (the identity of &; its value varies depending on the analysis).

$$r \quad \in \quad \{\bot,\ ID_{\&}\}$$

Demand environments

$$\theta \quad ::= \quad \langle \phi, r \rangle$$

$$\theta(x) \quad = \quad \phi(x), \qquad x \in dom(\phi)$$
$$= \quad r, \qquad \text{otherwise}$$
where $\langle \phi, r \rangle = \theta$

$$\theta_1 \ \sqcup \ \theta_2 \quad = \quad \langle \phi, r_1 \sqcup r_2 \rangle$$
where
$$\langle \phi_1, r_1 \rangle \quad = \quad \theta_1$$
$$\langle \phi_2, r_2 \rangle \quad = \quad \theta_2$$
$$\phi \quad = \quad [\, x \mapsto (\theta_1(x) \ \sqcup \ \theta_2(x))$$
$$\mid x \in dom(\theta_1) \cup dom(\theta_2) \,]$$

$$\theta_1 \ \& \ \theta_2 \quad = \quad \langle \phi, r_1 \& r_2 \rangle$$
where
$$\langle \phi_1, r_1 \rangle \quad = \quad \theta_1$$
$$\langle \phi_2, r_2 \rangle \quad = \quad \theta_2$$
$$\phi \quad = \quad [\, x \mapsto (\theta_1(x) \ \& \ \theta_2(x))$$
$$\mid x \in dom(\theta_1) \cup dom(\theta_2) \,]$$

Figure 5: Demand signatures and demand types

The *demand environment*, θ, consists of a pair $\langle \phi, r \rangle$, where ϕ is a finite mapping from variables to demands, and r is a demand. The lookup operation $\theta(x)$ gives the demand placed on a variable x by the demand environment θ. This operation is given in Figure 5: if x is in the domain of the finite mapping ϕ, then the result is simply $\phi(x)$; otherwise the result is r:

$$\theta(x) \quad = \quad \phi(x), \qquad x \in dom(\phi)$$
$$= \quad r, \qquad \text{otherwise}$$
where $\langle \phi, r \rangle = \theta$

In a demand environment $\theta = \langle \phi, r \rangle$, the result demand r is the demand placed on a variable that is not in the domain of θ. It can take one of two values (Figure 5):

- For terminating functions, r is $ID_{\&}$, the identity of the "&" operator.
- For diverging functions, r takes the value $\bot$, the identity of $\sqcup$.

The $\sqcup$ and & operations on demand environments are also given in Figure 5; they are defined so that

$$(\theta_1 \sqcup \theta_2)(x) = \theta_1(x) \ \sqcup \ \theta_2(x)$$

and similarly for &.

Because functions may have functions as results, a function may be applied to more arguments than its arity, and so we may find that an expression whose demand type has no argument demands is nevertheless applied to an argument. To accommodate this we use the following equivalence:

$$\theta \Rightarrow [\,] \quad \equiv \quad \theta \Rightarrow [\top]$$

That is, "extra" arguments are given the topmost demand $\top$. In the strictness lattice the topmost demand is L, but in other analyses it will differ.

Demand types are ordered in the obvious way, which we will need when taking fixpoints. *SLPJ: Is it obvious enough?*

Two particular demand types are also defined in Figure 5, namely *AbsType* and *BotType*. These are, respectively, the demand types of a completely unknown value, and of a divergent value.

For notational brevity, we sometimes omit the result demand from θ if it is $ID_{\&}$, and the mapping ϕ if it is empty. For example, we may write simply $[\mathtt{x} \mapsto S]$ instead of $\langle [\mathtt{x} \mapsto S], L \rangle$. If both are omitted, we may omit the "$\Rightarrow$" from a demand type. Finally, we often allow ourselves the informality of omitting the arity when the number of demands in the demand type is equal to the arity. For example, when we say that `seq` has demand signature SS, we really mean that it has signature

$$\langle 2,\ \langle [\,], L \rangle \Rightarrow SS \rangle$$

6 The strictness analysis

We are now ready to present the strictness analyser itself. A typical call to the analyser looks like this:

$$dt = \mathcal{S}[\![e]\!]\ \rho\ s$$

The analysis function $\mathcal{S}[\![e]\!]$ takes an expression e, together with an environment ρ and a demand s. The call returns a demand type dt which describes the demands that e places on its context when evaluated with demand s. The environment ρ gives the demand signatures for `let` or `letrec`-bound values that are in scope.

6.1 An example

The full definition of the analyser is given in Figure 6, but we begin with an example to give the idea. Consider the following call of the analyser:

$$\mathcal{S}[\![\texttt{(\textbackslash x -> x+y) z}]\!]\ \rho\ S$$

When the analyser sees an application, it analyses the function part of the application with a call demand, in this case $S(S)$:

$$\mathcal{S}[\![\texttt{\textbackslash x -> x+y}]\!]\ \rho\ S(S)$$

When the analyser sees a lambda, and its demand argument is a call demand, it analyses the body of the lambda with the subordinate demand, in this case S:

$$\mathcal{S}[\![\texttt{x+y}]\!]\ \rho\ S$$

The result of this call (skipping a few steps) is the demand type:

$$\mathcal{S}[\![\texttt{x+y}]\!]\ \rho\ S = [\texttt{x} \mapsto \texttt{S}, \texttt{y} \mapsto \texttt{S}] \Rightarrow [\,]$$

Now the analyser can complete its handling of the lambda, by removing the bound variable `x` from the free-variable set, and adding it to the argument list, thus:

$$\mathcal{S}[\![\texttt{\textbackslash x -> x+y}]\!]\ \rho\ S(S) = [\texttt{y} \mapsto \texttt{S}] \Rightarrow [S]$$

The analyser is now back to the original application, and now it can analyse the argument of the application (`z` in this case) using the first argument demand from the function's demand type (S in this case). So it performs the call

$$\mathcal{S}[\![\texttt{z}]\!]\ \rho\ S = [\texttt{z} \mapsto \texttt{S}] \Rightarrow [\,]$$

Finally, it completes the application by combining the demand types from the function part and the argument part:

$$\mathcal{S}[\![\texttt{(\textbackslash x -> x+y) z}]\!]\ \rho\ S(S) = [\texttt{y} \mapsto \texttt{S}, \texttt{z} \mapsto \texttt{S}] \Rightarrow []$$

6.2 Demand types and demand signatures

The whole analysis process is somewhat like type inference, which is why we use the term "demand type". It is very important, however, to distinguish a *demand type* from a *demand signature*.

- The environment, ρ, maps each in-scope `let` or `letrec`-bound variable to its *demand signature*.
- A demand signature is a simple encoding of a *demand transformer* (§5.1).
- A demand transformer transforms a *demand* into a *demand type*.
- A demand type gives the demands placed by a subexpression on its context; that is, both on its arguments and its free variables.

The function $\mathcal{S}[\![e]\!]\rho$ is itself a demand transformer: it takes a demand to a demand type. The demand signature for a function `f = rhs` is simply a crude summary of the function $\mathcal{S}[\![\texttt{rhs}]\!]\rho$, as we discussed in §5.1.

6.3 The analysis function

We are now ready to give the formal presentation of the strictness analyser in Figure 6.

If an expression e appears with lazy demand (L), then it has no argument demands and imposes demand L on all its free variables and on any other variables, so its demand type is *AbsType*.

So below we assume that the demand on the expression is strict, that is, of the form $\bot$ $(= S(\bot \ldots \bot))$ or $S(\overline{s})$.

The second equation deals with a variable x. The basic idea is that we look up `x`'s demand signature $\rho(x)$ in the environment, and use it as a demand transformer to transform the incoming demand s. We use an auxiliary function $\mathcal{DT}(\rho(x),\ s)$ to implement the demand transformation, which we discuss shortly. Finally, we must remember to record the demand s on x itself.

The third equation deals with the case of a product constructor, P, also by using an auxiliary function $\mathcal{PT}(P,\ s)$ for the demand transformer of the constructor.

We have already discussed the next equation, that for function applications, in §6.1. The demand placed on e_2 is the first argument demand s_a obtained from the demand type of e_1. If its argument demands are empty, the latter can be expanded to the form $\theta \Rightarrow s_a : \sigma$ (where ':' prepends an element to a sequence) using the demand type equivalences of Figure 5.

Notice that the returned argument demands, σ_1, come exclusively from the function (minus the first demand, of course); the argument demands from e_2 are discarded. We need the 'both' operation to combine the environments, θ_1 and θ_2, returned by analysing the function and its argument. This operation on demand environments is defined in Figure 5.

Dual to application, the equation for lambda finds the demand type for the body, extracts the demand on the bound variable x from the free-variable demands θ, and uses that to augment the returned argument demands.

`case` expressions that scrutinise a product constructor are dealt with by the next equation. The interesting point is that the demand on the case scrutinee, e is the product demand $S(\theta_a(x), \theta_a(y))$, where the sub-demands are obtained by seeing how the case alternative a consumes the components of the product, x and y.

All other forms of `case` expression are dealt with in the conventional way: just take the least upper bound of the alternatives of the `case`.

The equation for `letrec` uses the approach sketched in §5.1. We analyse the right hand side using a "vanilla demand" $S^n(S)$, where n is the arity of the right hand side. Then from the demand type returned by analysing the right hand side we build a demand signature, $\langle n,\ dt\rangle$. Finally, we analyse the body of the `letrec` in the extended environment ρ'. We compute the arity of the right hand side crudely, by simply counting lambdas, but *the correctness of the analysis does not depend on computing the "correct" arity* (whatever that is). A bad choice of arity will simply lead to less accurate results, because the demand transformer encoding will be less effective.

If the `letrec` is actually recursive, we must analyse the right hand side in the extended environment too. We follow the usual approach of computing a sequence of approximations to the demand signature for x, starting with the most aggressive one, $\bot = \langle 0,\ BotType\rangle$. Since the $\sqcup$ and & operators are monotonic and the lattice of demand signatures has finite height for any well-typed program, the limit can be computed in finitely many iterations. Notice that we

$\mathcal{S}[\![e]\!]\ \rho\ d$ takes an expression e, a demand environment ρ and an evaluation demand s, and computes a demand type dt for e.

$$
\begin{array}{rcl}
\mathcal{S}[\![e]\!] : StrEnv \to StrDmd \to DemandType & & \\
\rho : StrEnv & = & Var \to DmdSig \\
\\
\mathcal{S}[\![e]\!]\ \rho\ L & = & AbsType \\
\\
\mathcal{S}[\![x]\!]\ \rho\ s & = & let\ \ \theta \Rightarrow \sigma \ = \ \mathcal{DT}(\rho(x),\ s) \quad \text{if } x \in dom(\rho) \\
 & & \qquad\qquad\qquad\ \ = \ AbsType \qquad\quad \text{otherwise} \\
 & & in\ (\theta\ \&\ [\mathtt{x} \mapsto s]) \Rightarrow \sigma \\
\\
\mathcal{S}[\![P]\!]\ \rho\ s & = & \mathcal{PT}(P,\ s), \quad \text{where } P \text{ is a product constructor} \\
\\
\mathcal{S}[\![e_1\ e_2]\!]\ \rho\ s & = & let\ \ \theta_1 \Rightarrow s_a : \sigma_1 \ = \ \mathcal{S}[\![e_1]\!]\ \rho\ S(s) \\
 & & \qquad\qquad\ \ \theta_2 \Rightarrow \sigma_2 \ = \ \mathcal{S}[\![e_2]\!]\ \rho\ s_a \\
 & & in\ (\theta_1 \& \theta_2) \Rightarrow \sigma_1 \\
\\
\mathcal{S}[\![\backslash x.e]\!]\ \rho\ S(s) & = & let\ \ \theta \Rightarrow \sigma \ = \ \mathcal{S}[\![e]\!]\ \rho\ s \\
 & & \qquad\qquad\ \ s_x \ = \ \theta(x) \\
 & & in\ \theta \backslash \{x\} \Rightarrow s_x : \sigma \\
\\
\mathcal{S}[\![\mathtt{case}\ e\ \mathtt{of}\ (x,y)\ \mathtt{->}\ a]\!]\ \rho\ s & = & let\ \ \theta_a \Rightarrow \sigma_a \ = \ \mathcal{S}[\![a]\!]\ \rho\ s \\
 & & \qquad\qquad\ \ \theta_e \Rightarrow [\,] \ = \ \mathcal{S}[\![e]\!]\ \rho\ S(\theta_a(x),\ \theta_a(y)) \\
 & & in\ (\theta_a \backslash \{x,y\}) \& \theta_e \Rightarrow \sigma_a \\
\\
\mathcal{S}[\![\mathtt{case}\ e\ \mathtt{of}\ \overline{p_i\ \mathtt{->}\ a_i}]\!]\ \rho\ s & = & let\ \ \overline{\theta_i \Rightarrow \sigma_i} \ = \ \overline{\mathcal{S}[\![a_i]\!]\ \rho\ s} \\
 & & \qquad\qquad\ \ \theta_a \Rightarrow \sigma_a \ = \ (\bigsqcup \overline{\theta_i \backslash fv(p_i)}) \Rightarrow \bigsqcup \overline{\sigma_i} \\
 & & \qquad\qquad\ \ \theta_e \Rightarrow \sigma_e \ = \ \mathcal{S}[\![e]\!]\ \rho\ S \\
 & & in\ \theta_a \& \theta_e \Rightarrow \sigma_a \\
\\
\mathcal{S}[\![\mathtt{letrec}\ x\ \mathtt{=}\ e\ \mathtt{in}\ b]\!]\ \rho\ s & = & letrec \qquad n \ = \ arity(e) \\
 & & \qquad\qquad\ \ \theta \Rightarrow \sigma \ = \ \mathcal{S}[\![e]\!]\ \rho'\ S^n(S) \\
 & & \qquad\qquad\ \ \rho' \ = \ \rho[x \mapsto \langle n,\ \theta \backslash \{x\} \Rightarrow \sigma \rangle] \\
 & & \qquad\qquad\ \ \theta' \Rightarrow \sigma' \ = \ \mathcal{S}[\![b]\!]\ \rho'\ s \\
 & & in\ \theta' \backslash \{x\} \Rightarrow \sigma'
\end{array}
$$

$\mathcal{DT}(sig,\ s)$ takes a demand signature sig and a demand s, and applies the demand transformer described by sig to s, returning a demand.

$$
\begin{array}{rcll}
\mathcal{DT}(\langle n,\ dt \rangle,\ S^m(s)) & = & dt & \text{if } n \leq m \\
 & = & AbsType & \text{otherwise}
\end{array}
$$

$\mathcal{PT}(P,\ s)$ takes a product constructor P and a demand s, returns the constructor's demand type given that demand.

$$
\begin{array}{rcll}
\mathcal{PT}(P,\ S^m(S(s_1, \ldots, s_n))) & = & \langle [\,], L \rangle \Rightarrow [s_1, \ldots, s_n] & \text{if } n = m = arity(P) \\
 & = & AbsType & \text{otherwise}
\end{array}
$$

Figure 6: The evaluation demand analysis

take the fixpoint in the lattice of demand *types*, not in the (excessively rich) lattice of demand *transformers*.

Finally, we return to the demand-transformer function, $\mathcal{DT}(sig,\ s)$, also shown in Figure 6. It takes a demand signature and a demand, and returns a demand type. There are two cases to consider:

Enough arguments. If the incoming demand s is of the form $S(\ldots S(s)) = S^n(s)$, the nesting depth, n, of the call demands says how many arguments x is applied to. If n is at least as big as the number of arguments encoded in the demand signature, then we simply unleash the demand signature as a demand type.

Too few arguments. If there are too few arguments, we return *AbsType*, which places a lazy demand on all arguments and free variables.

The demand transformer for a product constructor, $\mathcal{PT}(P,\ s)$ is similar: provided the constructor is saturated, we can unleash the component demands of the incoming demand, $s_1, \ldots, s_n$, on the arguments of the constructor. Otherwise we just return the top demand type.

This completes the description of our strictness analysis. Compared to previous backwards strictness analyses, the main new elements are

- simple and cheap extension to handle higher-order functions via call demands,
- more accurate analysis of `let`-bindings via free-variable demands,
- and approximation of demand transformers by demand signatures, to make fixpointing fast and the analysis modular.

7 Absence analysis

Our next step is to apply exactly the same analysis framework to determine *absence*. For example, in the definition:

```
f x y = if x==0 then f (x-1) y
                else x
```

it is clear that `x` is used; it is slightly less obvious that `y` is not. Programmers seldom pass arguments that are entirely unused, but they often pass arguments that are only *partly* used. For example:

```
fst p = case p of { (x,y) -> x }
```

Here, the second component of the pair is unused. These two examples make it clear that absence analysis entails more than simply computing free variables.

7.1 Domains

The domain we use for absence analysis, its identities, and the type-indexed function that gives the domain for each type, are given in Figure 7.

The two elements that are common to every domain are:

The set $AbsDmd$ is the set of absence demands a defined thus:

a	::=	A	Definitely unused
	|	U	May be used
	|	$U(\overline{a})$	Product or function is used, components $\overline{s}$
$\overline{a}$	::=	$[a_1, \ldots, a_n]$	Tuple components

Identities on absence demands

$$U(U \ldots U) = U$$

Ordering on demands

$$\begin{array}{rcll} A & \sqsubseteq & U & \\ U(\overline{a_1}) & \sqsubseteq & U(\overline{a_2}) & \text{if } \overline{a_1} \sqsubseteq \overline{a_2} \end{array}$$

The $\sqcup$ and & operators

$$\begin{array}{rcll} A & \sqcup & a & = a \\ U(\overline{a_1}) & \sqcup & U(\overline{a_2}) & = U(\overline{a_1} \sqcup \overline{a_2}) \\ \\ a_1 & \& & a_2 & = a_1 \sqcup a_2 \end{array}$$

The absence domain for each type

$$\begin{array}{rcll} \mathcal{T}_a[\![\texttt{Int\#}]\!] & = & \{A, U\} & \text{Unlifted integers} \\ \mathcal{T}_a[\![(t_1\texttt{,}t_2)]\!] & = & U(\mathcal{T}_a[\![t_1]\!], \mathcal{T}_a[\![t_2]\!]) & \text{Product types} \\ & \cup & \{A\} & \\ \mathcal{T}_a[\![\texttt{[t]}]\!] & = & \{A, U\} & \text{Sum types} \\ \mathcal{T}_a[\![t_1\texttt{->}t_2]\!] & = & U(\mathcal{T}_s[\![t_2]\!]) & \text{Functions} \\ & \cup & \{A\} & \\ \mathcal{T}_a[\![\alpha]\!] & = & \{A, U\} & \text{Unknown types} \end{array}$$

Figure 7: Absence-demand domains, operators, and identies

A **(absent):** the value is not used at all, on any execution path. This is the bottom element of each domain, and we will write either A or $\perp$ interchangeably.

U **(used):** the value is used on some execution path. This is the top element of each domain.

The domain for unboxed types, such as `Int#`, has just these two points, as do the domains for sum types and polymorphic types. (In the case of strictness analysis, we needed three points for the latter domains.) The product domain is constructed in a similar way as for strictness, and there is an identify $U(UU) = U$ analogous to the one for strictness $S(LL) = S$. So we can calculate the domain for `Int` thus:

$$\begin{array}{rcl} \mathcal{T}_a[\![\texttt{Int}]\!] & = & S(\mathcal{T}_a[\![\texttt{Int\#}]\!]) \cup \{A\} \\ & = & \{U(A), U(U), A\} \\ & = & \{U(A), U, A\} \end{array}$$

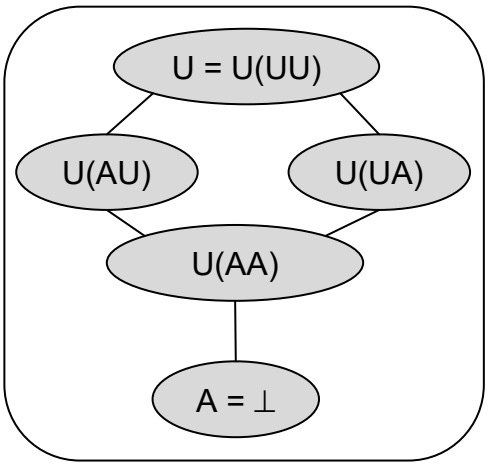

Figure 8: Absence for (integer) pairs

The demand A indicates that the integer is not used, U that it may be, and $U(A)$ that the *box* of the integer may be used, but the value will not be. *John: Weird! Can we give an example of a program which actually uses an integer that way??* The domain for pairs can be calculated similarly, and is depicted in Figure 8.

As Figure 7 shows, a significant difference between strictness and absence analysis is that the & is identical to $\sqcup$. Why? Consider the demands on `x` from the right hand sides of these two functions:

```
f1 x y = x + y
f2 x y = if x==0 then x else y
```

In `f1` we will use & to combine the demands on `x` from the arguments to `+`, while in `f2` we will use $\sqcup$ to combine demands from the branches of the `if`. For absence analysis there is no difference in these two operators: `x` is not absent in either `f1` or `f2`. Strictness analysis establishes that something is evaluated in *every* execution path, so there is a difference between combining information from parts of the same path (&) and from different paths ($\sqcup$). Absence analysis establishes that something might be used in *some* path: if we find a use, then that is all that matters; it's unimportant whether we combine information from alternative paths or parts of the same one.

7.2 Demand types

All the definitions concerning demand signatures and demand types (Figure 5) hold unchanged. There is an interesting point about return demands, though. Recall that a return demand r is drawn from the set

$$r \in \{\bot, ID_{\&}\}$$

For absence analysis, the identity of & is $\bot$, so it follows that r is always $\bot$ $(= A)$. So given a demand environment $\theta = \langle \rho, A \rangle$, any variable x not in the domain of ρ will be mapped to A – which is exactly as it should be. Absence analysis has no need to model divergence.

The function `error` has an interesting demand type:

$$\texttt{error} : \langle [\,], A \rangle \Rightarrow [U]$$

Note the $[U]$ part; it says that `error` uses its first argument (to print out the error message).

7.3 The analysis

With these preliminaries, the analysis function of Figure 6 works almost entirely unchanged. We have to make only the following adjustments: replace S by U, and L by A.

SLPJ: Again, rather an abrupt conclusion; what more should we say?

8 Combining strictness and absence

In this section we take the cartesian product of the two domains, approximate a bit, and end up with an interesting modification of the traditional 4-point domain [**?**]; only now we see where it comes from.

8.1 The demand domain

To compute the demands for the joint analysis we simply take the cartesian product of the strictness and absence domains. Figure 9 gives the syntax of joint demands sa, and for each it gives the corresponding pair $\langle s, a \rangle$ from the product lattice, where s is a strictness demand and a is an absence demand.

As before, the demand domain is indexed by type. To begin with, consider the simplest demand domain, that for lists and type variables. For these types, the strictness domain is just $\{\bot, S, L\}$, and the absence domain is $\{A, U\}$ (see Figures 2 and 7 respectively). We can make sense of the following joint demands placed by an expression e on a variable x:

$Lazy = \langle L, U \rangle$. x is mentioned (U), but not necessarily evaluated (L).

$Abs = \langle L, A \rangle$. x is not mentioned at all (A), and (unsurprisingly) is not evaluated (L).

$Str = \langle S, U \rangle$. x is both mentioned and evaluated to at least head-normal form.

$Err = \langle \bot, U \rangle$. x is mentioned, and e diverges.

$\bot = \langle \bot, A \rangle$. x is not mentioned, and e diverges.

In each case we have invented a new joint demand (e.g. Err) to name the demand pair (e.g. $\langle \bot, U \rangle$). What about the missing point $\langle S, A \rangle$? The S means "every terminating path evaluates x", while the A means "no path evaluates x". So the joint demand $\langle S, A \rangle$ means that no path terminates, and no path evaluates x, so it is the same as $\langle \bot, A \rangle$. *SLPJ: Idea! Can we prove this claim in the section on projections? That would be convincing.*

Their ordering relationship is given by:

$$\langle s_1, d_1 \rangle \sqsubseteq \langle s_2, d_2 \rangle \equiv s_1 \sqsubseteq s_2 \wedge d_1 \sqsubseteq d_2$$

Figure 10 shows these ordering relationships, and already we can see that matters are more complicated than the four-point domain we find in the literature. In particular, the distinction between Err and $\bot$ is interesting. Consider:

Joint demands

$$
\begin{array}{rcll}
sa & ::= & Lazy & = \langle L,\, U\rangle \\
 & | & Lazy(\overline{sa}) & = \langle L,\, U(\overline{a})\rangle \\
 & | & Str & = \langle S,\, U\rangle \\
 & | & Str(\overline{sa}) & = \langle S(\overline{s}),\, U(\overline{a})\rangle \\
 & | & Abs & = \langle L,\, A\rangle \\
 & | & Err & = \langle \bot,\, U\rangle \\
 & | & \bot & = \langle \bot,\, A\rangle
\end{array}
$$

Identities on demands

$$
\begin{array}{rcl}
Str(\ldots\bot\ldots) & = & Err \\
Str(Lazy\ldots Lazy) & = & Str \\
Lazy(Lazy\ldots Lazy) & = & Lazy
\end{array}
$$

Ordering on demands

$$
\begin{array}{rcll}
\bot & \sqsubseteq & sa & \\
sa & \sqsubseteq & Lazy & \\
Str(\overline{sa_1}) & \sqsubseteq & Str(\overline{sa_2}) & \text{if } \overline{sa_1} \sqsubseteq \overline{sa_2}
\end{array}
$$

Operations on demands

$$
\begin{array}{rcll}
\bot & \sqcup & sa & = sa \\
Abs & \sqcup & \bot & = Abs \\
Abs & \sqcup & sa & = Lazy \\
Str(\overline{sa}) & \sqcup & \bot & = Str(\overline{sa}) \\
Str(\overline{sa}) & \sqcup & Abs & = Lazy(\overline{sa}) \\
Str(\overline{sa}) & \sqcup & Err & = Str \\
Str(\overline{sa_1}) & \sqcup & Str(\overline{sa_1}) & = Str(\overline{sa_1} \sqcup \overline{sa_2}) \\
Str(\overline{sa_1}) & \sqcup & Lazy(\overline{sa_1}) & = Lazy(\overline{sa_1} \sqcup \overline{sa_2}) \\
Lazy(\overline{sa}) & \sqcup & \bot & = Lazy(\overline{sa}) \\
Lazy(\overline{sa}) & \sqcup & Abs & = Lazy(\overline{sa}) \\
Lazy(\overline{sa}) & \sqcup & Err & = Lazy \\
Lazy(\overline{sa}) & \sqcup & Str(\overline{sa_1}) & = Lazy(\overline{sa_1} \sqcup \overline{sa_2}) \\
Lazy(\overline{sa}) & \sqcup & Lazy(\overline{sa_1}) & = Lazy(\overline{sa_1} \sqcup \overline{sa_2})
\end{array}
$$

$$
\begin{array}{rcll}
\bot & \& & \bot & = \bot \\
\bot & \& & Abs & = \bot \\
\bot & \& & sa & = Err \\
Abs & \& & sa & = sa \\
Str(\overline{sa}) & \& & \bot & = \bot \\
Str(\overline{sa}) & \& & Err & = Err \\
Str(\overline{sa}) & \& & Abs & = Str(\overline{sa}) \\
Str(\overline{sa_1}) & \& & Str(\overline{sa_1}) & = Str(\overline{sa_1}\&\overline{sa_2}) \\
Str(\overline{sa}) & \& & Lazy(\overline{sa_1}) & = Str(\overline{sa_1}\&\overline{sa_2}) \\
Lazy(\overline{sa}) & \& & \bot & = Err \\
Lazy(\overline{sa}) & \& & Err & = Err \\
Lazy(\overline{sa}) & \& & Abs & = Lazy(\overline{sa}) \\
Lazy(\overline{sa}) & \& & sa & = sa
\end{array}
$$

Figure 9: Joint strictness and absence

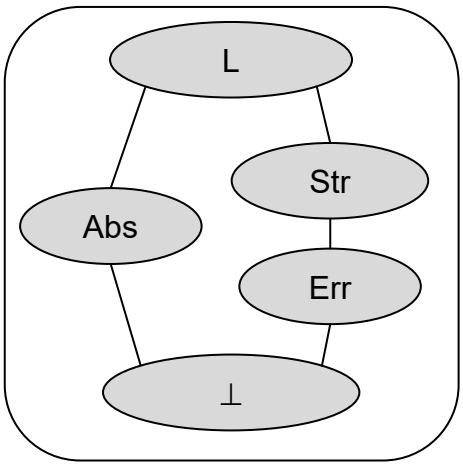

Figure 10: Joint-demand lattice for lists

```
f x y = error ("Urk" ++ x)
```

The demand on `x` is Err, and on `y` is $\bot$. This distinction is truly useful: we must pass `x` to `f`, but we need not pass `y`.

John: Hmm. Why don't we allow $Err(\overline{sa})$? It has a natural meaning and it seems useful. What do you think, Simon?

8.2 Products and functions

Matters become more interesting when we consider structured types. Even `Int` is more complicated than one would think. For `Int`, the demands we are interested in are:

$$
\begin{array}{l}
\{\langle s,\, a\rangle | s \in \mathcal{T}_s[\![\texttt{Int}]\!], a \in \mathcal{T}_a[\![\texttt{Int}]\!]\} \\
= \{\langle s,\, a\rangle | s \in \{\bot, S, L\}, a \in \{A, U(A), U(U)\}\}
\end{array}
$$

Now we do some approximation. We name the following joint demands (Figure 9):

$$
\begin{array}{rcl}
Lazy & = & \langle L,\, U\rangle \\
Lazy(Abs) & = & \langle L,\, U(A)\rangle \\
Str & = & \langle S,\, U\rangle \\
Str(Abs) & = & \langle S,\, U(A)\rangle \\
Abs & = & \langle L,\, A\rangle \\
Err & = & \langle \bot,\, U\rangle \\
\bot & = & \langle \bot,\, A\rangle
\end{array}
$$

There are two elements missing from this list. We identify $\langle S,\, A\rangle$ with $\bot$ as before. But what about $\langle \bot,\, U(A)\rangle$? That demand would be created by a function like this:

```
f (I# x) = error "Urk"
```

From a strictness point of view, `f` is hyper-strict in its argument (because of the call to `error`), but in fact the payload of the `Int`, namely `x`, is not used. We choose not to record information at this level of precision, by not having an element of the joint domain for $\langle \bot,\, U(A)\rangle$. Whenever we might want it, we can approximate upwards to get Err instead.

Another very plausible approximation would be to approximate $\langle L,\, U(\overline{a})\rangle$ to $Lazy$ for any absence demands $\overline{a}$. Why do we instead choose to preserve absence information even for a lazy demand? After all, consider:

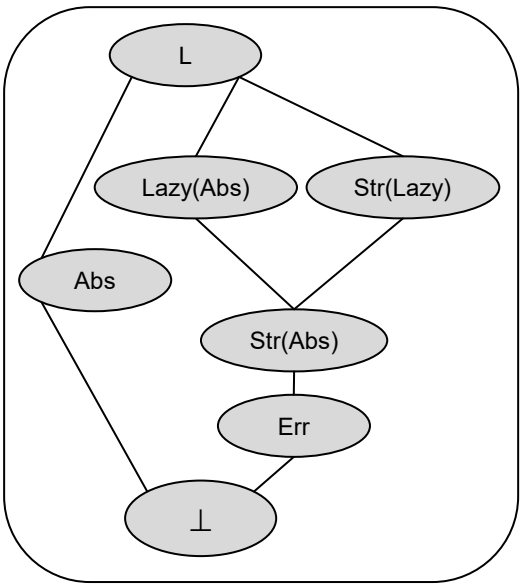

Figure 11: Joint-demand lattice for `Int`

```
f :: (a,b) -> Maybe a
f x = Just (fst x)
```

This function places demand $Lazy(Lazy, Abs)$ on `x`, and so `f` has no useful worker/wrapper split: we must pass `x` entire to `f`. Thus motivated, our first implementation did indeed approximate $\langle L,\, a\rangle$ to $Lazy$. However, consider this function:

```
f x = if fst x then
         Nothing
      else
         Just (fst x)
```

If the demand from `Just (fst x)` is simply $Lazy$, then the demand on `x` for the entire function will be $Str = Str(Str, Lazy)$. But that is bad! It is plain as a pikestaff that the demand on `x` should be $Str(Str, Abs)$; that is, the second component of `x` is not used, and should not be passed from the wrapper to the worker. If, instead, we do less approximation, we get the demand $Lazy(Str, Abs)$ from `Just (fst x)`, and $Str(Str, Abs)$ from the `if` condition; and

$$Lazy(Str, Abs)\ \&\ Str(Str, Abs) = Str(Str, Abs)$$

which is what we want. In short, even a lazy demand should record which parts of the value will not be used.

Figure 11 shows the joint-demand lattice for `Int`. At this point, the merit of our modular approach becomes clear. The strictness and absence domains were relatively easy to define, but the joint domain even for a simple type like `Int` has become rather complicated. By taking the product of strictness and absence domains we have a systematic approach to constructing the joint domain.

8.3 Operations over demands

Figure 9 give the $\sqcup$ and & operations over demands. We do not give cases for Str and Err because they can be expanded using the identities. Again, these operations can be calculated from the operations over strictness and absence demands, using the guide:

$$\begin{array}{rcl} \langle s_1 \sqcup s_2,\, a_1 \sqcup a_2\rangle & \sqsubseteq & \langle s_1,\, a_1\rangle \sqcup \langle s_2,\, a_2\rangle \\ \langle s_1 \& s_2,\, a_1 \& a_2\rangle & \sqsubseteq & \langle s_1,\, a_1\rangle \& \langle s_2,\, a_2\rangle \end{array}$$

For example, we can compute $Str(\overline{sa})\ \&\ Abs$ thus:

$$\begin{array}{rcl} Str(\overline{sa})\ \&\ Abs & = & \langle S(\overline{s}),\, U(\overline{a})\rangle\ \&\ \langle L,\, A\rangle \\ & = & \langle S(\overline{s}) \& L,\, U(\overline{a}) \& A\rangle \\ & = & \langle S(\overline{s}),\, U(\overline{a})\rangle \\ & = & Str(\overline{sa}) \end{array}$$

Notice that "&" is neither least upper bound nor greatest lower bound in the combined lattice.

8.4 Demand signatures and demand types

The same definitions for demand signatures and demand types hold as before. We only need to identify the values of r, the return demand. Recall from Figure 5 that a return demand is an element of:

$$r \in \{\bot,\, ID_\&\}$$

In our joint analysis, the identity of & is $\langle ID_\&,\, ID_\&\rangle = \langle L,\, A\rangle = Abs$. So we calculate that for joint strictness/absence analysis:

$$r \in \{\bot,\, Abs\}$$

8.5 The analysis

We exploit the following equation:

$$\mathcal{S}[\![e]\!]\ \rho\ (sa_1 \sqcup sa_2) \;=\; (\mathcal{S}[\![e]\!]\ \rho\ sa_1) \sqcup (\mathcal{S}[\![e]\!]\ \rho\ sa_2)$$

SLPJ: Proof?

There are two new features.

- The very first equation becomes:
- Abstracting over $Lazy(Str, Abs)$ gives just $Lazy$.

John: This section is highly incomplete!

9 Practical issues

Next, we turn our attention to some issues that turn out to be important in practice, principally to do with fixpoints. These issues never occurred to us before we began, but they are crucial to good practical performance.

9.1 Returning an annotated expression

In our implementation, the demand analyser returns not only a demand type, but also an annotated expression, in which:

- Each let(rec) binder is annotated with its demand signature.

- Each binder (lambda, case, and let(rec)) is annotated with the demand placed on it if the expression is evaluated at all.

The former information is used to drive the worker/wrapper split that follows. Both annotations are used during program transformation and code generation to transform call-by-name into call-by-value.

9.2 Finding fixpoints

As we have already remarked, finding fixpoints for nested recursive functions can be expensive. For example, consider the following Haskell function:

```
f xs = [y+1 | x <- xs, y <- h x]
```

GHC will turn the list comprehension (which really has two nested loops) into something like the following:

```
f [] = []
f (x:xs) = letrec
             g []     = f xs
             g (y:ys) = y+1 : g ys
           in
           g (h x)
```

The trouble is that the analyser must find a fixpoint for the inner function, `g`, on each iteration of the fixpoint finder for the outer function, `f`. If functions (or list comprehensions) are deeply nested, as can occur, this can lead to exponential behaviour, even if each fixpoint iteration converges after only two cycles.

While this remains the worst-case behaviour, there is a simple trick that dramatically improves the behaviour of common cases. It relies on the following observation. The iterations of the fixpoint process for `f` generates a monotonically increasing sequence of demand signatures for `f`. Therefore, each time we begin the fixpoint process for `g`, the environment contains values that are greater (in the demand lattice) than the corresponding values the previous time we encountered `g`. It follows that the correct fixpoint for `g` will be greater than the correct fixpoint found on the previous iteration of `f`. Therefore *we can begin the fixpoint process for* `g` *not with the bottom value, but rather with the result of the previous analysis.*

It is simple to implement this idea. Each iteration of the `f`'s fixpoint process yields a new right-hand side for `f`, as well as its demand type. We simply feed that new right hand side, whose binders are decorated with their demand signatures, into the next iteration. Then, when beginning the fixpoint process for `g`, we can start from the demand signature computed, conveniently attached to the binding occurrence of `g`.

In practice, most of the fixpoint processes of the inner function then converge in a single iteration, which prevents exponential behaviour.

This technique is fairly well-known as folk lore, but it was not written down until Henglein's paper [**?**]. (This paper is fairly dense, and the fact that it contains this extremely useful implementation hack may not be immediately apparent.)

SLPJ: The explanation is a bit armwavey; can it be improved? There should be some nunbers to back this up. John: I like it! Of course, numbers would be good. Analysis times on some benchmarks with and without the optimisation?

9.3 Splitting θ

In §5.3 we noted the importance of including information about free variables in the demand signature computed for a local definition. What we did not mention there is that doing so greatly enriches the lattice of demand signatures, and can therefore make convergence of fixpoints much slower. Indeed, we found this to be a real problem in practice. Even using the fixpoint technique described above, we still encountered exponential behaviour.

Some careful inspection of actual examples showed that the trouble really concerned variables that are used *lazily*. Consider:

```
f x y = let g z = if x then (y,z) else (z,y)
        in
        ...g...g...
```

The argument of §5.3 was that `g` should get a demand signature something like:

$$\mathtt{g} : \langle 1, \langle [\mathtt{x} \mapsto S, \mathtt{y} \mapsto L], L \rangle \Rightarrow L \rangle$$

The mapping in the demand signature says that a call of `g` places a strict demand on `x` and a lazy demand on `y`.

Something is gained by unleashing a strict demand on `x` at `g`'s call sites; we may get better overall strictness for `x` (§5.3). However, *nothing is gained by unleashing a lazy demand on* `y` *at* `g`*'s call sites.* Each call site (whether saturated or not) will unleash a lazy demand, and they will all combine to give an overall lazy demand (unless there is some other strict demand on `y`). It would be simpler and more direct, after analysing the right hand side of `g`, to give `g` the simpler signature:

$$\mathtt{g} : \langle 1, \langle [\mathtt{x} \mapsto S], L \rangle \Rightarrow L \rangle$$

and to derive a lazy demand on `y` from the *definition* of `g` (rather than from its call sites).

So the idea is this. After analysing the right hand side of a function, split the θ it returns into the variables with lazy demands and those with strict demands. Put only the strict demands into the demand signature for the function; the lazy ones can simply float outwards. More precisely, here is the revised `let(rec)` rule:

$$\begin{array}{lrcl}
\mathcal{S}[\![\texttt{letrec } x \texttt{ = } e \texttt{ in } b]\!]\ \rho\ s & & & \\
= \ \mathit{letrec} & n & = & \mathit{arity}(e) \\
& \theta \Rightarrow \sigma & = & \mathcal{S}[\![e]\!]\ \rho'\ \mathit{Str}^n(\mathit{Str}) \\
& (\theta_{lazy}, \theta_{str}) & = & \mathit{split}(\theta) \\
& \rho' & = & \rho[x \mapsto \langle n,\ \theta_{str} \Rightarrow \sigma \rangle] \\
& \theta' \Rightarrow \sigma' & = & \mathcal{S}[\![b]\!]\ \rho'\ s \\
\quad \mathit{in}\ \theta' \& \theta_{lazy} \Rightarrow \sigma' & & &
\end{array}$$

This refinement turns out to be devastatingly effective in practice. Several troublesome programs that took a huge number of fixpoint iterations before now converged in one or two.

SLPJ: Again, is this enough? Numbers needed here too. John: No, this is not clear. This discusses only strictness analysis — but what happens once absence is added? Is it Lazy demands that are factored out? Or is Abs also factored? What is the criterion in general? My guess is that

anything $\sqsupseteq$ Abs can be factored, but we should really do a proof that doing so does not change the result. Will think about this.

10 What does it all mean?

Backwards strictness-and-absence analysis has a nice theory developed by Wadler and Hughes [?], which we shall adapt to give a semantics to the demands and demand types in this paper.

10.1 Projections and absence analysis

In this section we will consider the semantics of *absence* demands, as described in Section 7, leaving strictness for the next section.

Wadler and Hughes' theory [?] is based on interpreting a demand as a *projection*: a function from a domain to itself that is idempotent and approximates the identity. That is,

$$p : D \to D \text{ is a projection if } p = p \circ p \text{ and } p \sqsubseteq id.$$

An example of a projection is the function which maps every value to $\bot$, and indeed, this is how we shall interpret A — it is the projection which discards its argument entirely.

$$A\ x = \bot$$

Now we can see that if a function f satisfies

$$f = f \circ A$$

then it must be a *constant* function — since, for all x, $fx = f\ \bot$. That is, it does not use its argument, or to put it another way, it places the demand A on its argument!

If we think of a projection as mapping a value to "the part that is used", replacing the unused parts by $\bot$, then it is clear that we should interpret the demand U (use the whole value) as the identity projection:

$$Ux = x$$

A function f may place the demand U on its argument if

$$f = f \circ U$$

All functions satisfy this condition, of course, which just says that every function "may use" its argument. In practice the best information is given by the *least* projection p such that

$$f = f \circ p$$

A static analyser cannot in general find the *least* such p, but we are satisfied if it usually finds a "small" one.

Demands on tuples are modelled by projections that apply other projections on the components separately. Thus (recalling that Haskell tuples are lifted)

$$\begin{array}{rcl} U(p_1,p_2)\ (x,y) & = & (p_1\ x, p_2\ y) \\ U(p_1,p_2)\ \bot & = & \bot \end{array}$$

Now it is straightforward to verify, for example, that

$$\texttt{fst} = \texttt{fst} \circ U(U,A)$$

that is, `fst` places the demand $U(U,A)$ on its argument.

The '&' operator is needed when function arguments are used more than once, for example in the definition

```
f x = g (x,x)
```

If `g` places the demand $U(p_1,p_2)$ on its argument, then `f` places the demand $p_1 \& p_2$, as we have seen. If we ask what this implies semantically, we see that if

$$\texttt{g} = \texttt{g} \circ U(p_1,p_2)$$

then it must be true that

$$\texttt{f} = \texttt{f} \circ (p_1 \& p_2)$$

Restating this in terms of `g`, whenever $\texttt{g} = \texttt{g} \circ U(p_1,p_2)$, then we must have

$$\texttt{g} = \texttt{g} \circ U(p_1\&p_2, p_1\&p_2)$$

This condition constrains how we may define '&' on projections. It will always hold, provided

$$U(p_1,p_2) \sqsubseteq U(p_1\&p_2, p_1\&p_2) \tag{1}$$

since then

$$\begin{array}{rcl} \texttt{g} \circ U(p_1\&p_2, p_1\&p_2) & \sqsupseteq & \texttt{g} \circ U(p_1,p_2) \\ & = & g \\ & \sqsupseteq & \texttt{g} \circ U(p_1\&p_2, p_1\&p_2) \end{array}$$

(since $U(p_1\&p_2, p_1\&p_2) \sqsubseteq id$). An obvious way to fulfill equation 1 is to take

$$p_1 \& p_2 \equiv p_1 \sqcup p_2$$

and indeed, as we saw above, for absence analysis '&' *is* the same as '$\sqcup$'. But we will use equation 1 again when we consider strictness analysis, and derive a better result.

So far we have assumed that the result of the function is consumed with demand U, but we can easily generalise the idea: if the function function f is called with a demand p on its result, we may say it places a demand q on its argument if

$$p \circ f = p \circ f \circ q$$

For example, given the definition

```
swap (x,y) = (y,x)
```

we can check that

$$U(U,A) \circ \texttt{swap} = U(U,A) \circ \texttt{swap} \circ U(A,U)$$

That is, if the second component of the result is not used, then neither is the first component of the argument — as expected.

This is the way Wadler and Hughes modelled absence — but does it correctly handle the awkward cases involving `error` discussed in section 2.2? Yes it does, provided we use Peyton-Jones et al's semantics of "imprecise exceptions" [?].

This semantics is designed to allow the compiler to make transformations that change the error that an erroneous program encounters, without thereby identifying all errors with $\bot$. In the semantics, erroneous programs denote a *set* of possible errors, not just one, and when the program is run the result is guaranteed only to be a member of the set. The sets are ordered using the Smyth powerdomain order, so that larger, less-precise sets approximate smaller, more-precise ones. Non-termination is then identified with the set of *all* possible errors — the largest, least-precise error set of all.

Now recall

```
g2 (x,y) = error x
```

which "diverges", but uses `x`. Clearly

$$\texttt{g2} = \texttt{g2} \circ U(U, A)$$

(that is, `y` is not used), but

$$\texttt{g2} \neq \texttt{g2} \circ U(A, A)$$

since when `g2` is applied to (x, y) it returns a *singleton* set of possible errors just containing x, while the right hand side returns $\bot$, the set of all possible errors. Thus we cannot say that `x` is not used.

10.2 Projections and strictness

The theory above models absence analysis nicely, but is not sufficient to model strictness analysis. Intuitively, the problem is that, having used $\bot$ to represent a missing value — something which will trigger divergence *if evaluated* — we cannot at the same time use it to represent divergence itself. Wadler and Hughes' solution was to add a *new* bottom element to the semantic domain, below the existing one (i.e. lifting the domain), with the new element representing divergence itself. To avoid confusion, the new bottom will be called $\searrow$ ("lightning bolt"). It is important to realise that we do *not* need to give a new semantics to Haskell, in which lightning bolts appear. We interpret Haskell programs as usual, but we model demands by *projections* on the semantic domain with one additional element, lightning bolt, rather than the original semantic domain. We will need to lift the semantics of Haskell functions to the extended domain, but given a function `f` this is easily done by taking `f` $\searrow = \searrow$. At every other point, `f` retains the usual semantics.

Now, in the extended domain we distinguish between a closure which will loop if evaluated ($\bot$), and non-termination itself $\searrow$. We can therefore model evaluating a closure as a projection:

$$\begin{array}{rcl} S\ \searrow & = & \searrow \\ S\ \bot & = & \searrow \\ S\ x & = & x, \text{ otherwise} \end{array}$$

We map an *unevaluated* looping closure ($\bot$) to true divergence ($\searrow$), while leaving terminating values unchanged. This is the projection that models strict demand; we can ask whether `f` uses its argument strictly when called in a strict context, just be asking whether

$$S \circ \texttt{f} = S \circ \texttt{f} \circ S$$

Applying both sides to $\bot$ we see

$$\begin{array}{rrcl} & (S \circ \texttt{f})\ \bot & = & (S \circ \texttt{f} \circ S)\ \bot \\ \Longrightarrow & S\ (\texttt{f}\ \bot) & = & S\ (\texttt{f}\ (S\ \bot)) \\ \Longrightarrow & S\ (\texttt{f}\ \bot) & = & S\ (\texttt{f}\ \searrow) \\ \Longrightarrow & S\ (\texttt{f}\ \bot) & = & S\ \searrow \\ \Longrightarrow & S\ (\texttt{f}\ \bot) & = & \searrow \\ \Longrightarrow & \texttt{f}\ \bot & = & \bot \end{array}$$

so this condition does indeed imply that `f` is strict. Similarly we can define strictness projections on tuples

$$\begin{array}{rcll} S(p_1, p_2)\ \searrow & = & \searrow & \\ S(p_1, p_2)\ \bot & = & \searrow & \\ S(p_1, p_2)\ (x, y) & = & \searrow, & \text{if } p_1\ x = \searrow \text{ or } p_2\ y = \searrow \\ & = & (p_1\ x, p_2\ y), & \text{otherwise} \end{array}$$

and finally, we model L (no evaluation) just by the identity function. Now we can pose questions "does `f` place demand q on its argument when called with demand p" by asking whether

$$p \circ \texttt{f} = p \circ \texttt{f} \circ q$$

as before.

How should the & operator be defined on the domain with lightning bolt? In the last section, we concluded that it must be defined so that

$$U(p_1, p_2) \sqsubseteq U(p_1 \& p_2, p_1 \& p_2)$$

In this section we have replaced $U(p_1, p_2)$ by $S(p_1, p_2)$, but we can make the same argument and conclude that & must satisfy

$$S(p_1, p_2) \sqsubseteq S(p_1 \& p_2, p_1 \& p_2)$$

However, we can now take advantage of the fact that $S(p_1, p_2)$ returns $\searrow$ if *either* p_1 or p_2 does, by defining

$$(p_1 \& p_2)\ x = \begin{cases} \searrow, & \text{if } p_1\ x = \searrow \text{ or } p_2\ x = \searrow \\ p_1\ x \sqcup p_2\ x, & \text{otherwise} \end{cases}$$

This is the semantic definition of &: it takes advantage of strictness in either operand, but if neither is strict behaves as $\sqcup$. Clearly this definition satisfies the necessary condition, and moreover we have that $p_1 \& p_2 \sqsubseteq p_1 \sqcup p_2$ and in general is different, so we obtain a more precise analysis.

The result of all this is a unified semantic framework for strictness and absence analysis. *SLPJ: Well, no one could argue with something as desirable as a "unified semantic treatment". Every schoolgirl should have one. But what the dickens is it, and what can one do with such a wonder? John: Promulgate Truth, Freedom, and the Functional Way, of course! OK, I take your point.*

10.3 Projections and function values

Wadler and Hughes' article was restricted to a first-order language. Numerous attempts were made to generalise the approach to higher-order languages [?, ?, ?], but these all involved departing more-or-less radically from the simple model of demands as projections. In this paper, on the other hand, we can model our analysis in a purely projection-based framework. We do need to adapt the theory a little to handle function values, though.

SLPJ: OK, this is a great statement. Sounds as if there is a qualitiative breakthrough here, yes? John: Yes, it's good! I'm going to reread some of the old projection papers, though, just to make sure the idea doesn't appear in some corner.

Firstly, we will need projections on function values. In this section we use the notation

$$(p \to q)\ f = q \circ f \circ p$$

to denote such projections. Clearly if p and q are projection on types σ and τ, then $p \to q$ is a projection on functions of type $\sigma \to \tau$. The call demands $S(p)$ used in the analysis can be modelled by function projections $ID \to p$: they impose a demand on the result of a function, but not on its argument.

Wadler and Hughes' *safety condition* for projection analysis,

$$p \circ \mathtt{f} = p \circ \mathtt{f} \circ q$$

can now be rephrased as

$$(ID \to p)\ \mathtt{f} = (q \to p)\ \mathtt{f}$$

or as

$$(ID \to p)\ \mathtt{f} = (ID \to p)\ ((q \to ID)\ \mathtt{f})$$

So, when this condition holds, if we see a use of `f` in a context where it is evaluated by $(ID \to p)$, then we can safely apply another projection $(q \to ID)$ to evaluate it a bit more.

More generally, if we know that e appears in a context $p'\ e$, then we can replace e by $q'\ e$ (thus discarding components or making evaluation stricter) provided that $p'\ e = p'\ (q'\ e)$. We will take this new form to be the safety condition for the analysis in this paper.

> **Safety condition:** Let the result of analysing an expression e with demand p' be the demand q'. Then we must have
>
> $$p'\ e = p'\ (q'\ e)$$

When e is a first order function, p' is $ID \to p$ (i.e. $S(p)$) and q' is $q \to ID$, then this condition reduces to Wadler and Hughes' original safety condition. But the new condition also applies directly to higher-order and curried functions.

For example, suppose `f` is a curried function of type $\sigma_1 \to \ldots \sigma_n \to \tau$, called in the context $S^n(p)$ (meaning p nested inside n applications of S). A projection q' produced by the analyser will be of the form $q_1 \to \ldots q_n \to ID$, satisfying

$$S^n(p)\ \mathtt{f} = S^n(p)\ ((q_1 \to \ldots q_n \to ID)\ \mathtt{f})$$

A call in a strict context, $p\ (\mathtt{f}\ e_1 \ldots e_n)$, can thus be replaced by

$$p\ (((q_1 \to \ldots q_n \to ID)\ \mathtt{f})\ e_1 \ldots e_n)$$

which is equal to

$$p\ (\mathtt{f}\ (q_1\ e_1) \ldots (q_n\ e_n))$$

That is, we can evaluate each e_i in the context q_i. We see that the list of argument demands $[q_1 \ldots q_n]$ in a demand type can thus be modelled semantically by the projection $q_1 \to \ldots q_n \to ID$.

SLPJ: Which in turn makes me wonder whether we shouldn't represent a demand type using curried arrows for the σ part instead of the $[s_1, ..s_n]$ notation? John: Yes... I like that idea!

SLPJ: I think we want some soundness statement like: if

$$\mathcal{S}[\![e]\!]\rho\ d = \theta \Rightarrow \sigma$$

then

$$d(\mathcal{E}(e)\rho) = d(\sigma(\mathcal{E}(e)\theta(\rho)))$$

That would help to set the overall context for the work of this section.

John: Yes! I agree completely.

10.4 The demand environment

Of course, in general the meaning of an expression — even a function — depends on the environment. The semantics of an expression is indeed a function from the environment to its value; the semantics of a function-valued expression is consequently a *curried* function from the environment and arguments to its result. When we analyse a Haskell expression of type $\sigma_1 \to \ldots \to \sigma_n \to \tau$, we are really analysing something whose semantics lies in the type $Env \to \sigma_1 \to \ldots \to \sigma_n \to \tau$. So we should expect the projection we obtain, according to the safety condition above, to be of the form

$$q_\rho \to q_1 \to \ldots q_n \to ID$$

where q_ρ is a projection on the environment. Of course, we represent such projections componentwise (by giving a projection for each name) – for example $[x \mapsto q_1, y \mapsto q_2]$. Thus we see that the demand environments that appear in demand types are also modelled naturally as a part of a projection on the semantics of expressions.

10.5 Result demands

What of result demands? To explain them semantically, we make a slight digression.

SLPJ: I think we could give this diression when talking of the demand environment, couldn't we? John: Well, the first part is just explaining that projections on environments correspond to part of demand types. This applies however we use environments. Then this next bit is about using a jolly strange semantics in order to get better results — I thought it well to separate them.

First of all, notice that every use of an identifier is (obviously) within the scope of its definition — and therefore, within the scope of every other identifier which was in scope at its definition. To put it another way, the environment at the point a variable is used is always strictly larger than the environment at the point the variable was bound. If a variable is bound by a **let** or **letrec** binding, then the value it is bound to is interpreted in the environment in scope at the definition. It follows that, *if there are no name clashes, we can equally well interpret the bodies of* **let***-bound variables in the scope where the variable is used*, instead of where it is defined. That is, dynamic binding is equivalent to static binding, provided there are no name clashes (and assuming that λ-expressions are closed in the environment where they appear, i.e. there is no "FUNARG problem"). GHC

eliminates name clashes in an earlier phase by renaming offending variables, so we are justified in assuming that they do not occur.

SLPJ: I kind of understand, but would it not be clearer to give a fragment of the denotational semantics thus: instead of

$$\mathcal{E}(letx = rinb)\rho = \mathcal{E}b\rho[x \mapsto \mathcal{E}(r)\rho]\mathcal{E}(x)\rho = \rho(x)$$

we have

$$\mathcal{E}(letx = rinb)\rho = \mathcal{E}b\rho[x \mapsto \mathcal{E}(r)]\mathcal{E}(x)\rho = \rho(x)\rho$$

Theorem: this change makes no difference. John: Yes, OK. That'll be an improvement.

So let us consider a semantics for Haskell in which **let** and **letrec** bindings bind names to *functions from the environment where they are used* to a value. As we have argued, the semantics is equivalent to the standard one, for name-clash-free programs. The advantage of using this semantics is that it lets us associate demands on the free variables of a **let** bound variable with the point where it is used, rather than the point where it is defined, which as we saw in section 5.3, can give us more accurate results. But now an occurrence of a variable denotes a function from a *possibly larger* environment to a value, so the result of analysis must give us a projection on this larger environment, not on the environment at the definition.

What projection, then, should we apply when x is used, to the values of variables which were not in scope at the definition of x? One safe possibility is to apply A to them: since the body of x cannot depend on them, this is certain to be safe. But *if x is undefined* then we can do better: since the result of evaluating x strictly will be $\searrow$ anyway, we might as well project the extra variables to $\searrow$ directly. That is, we can apply the least projection $\lambda z \rightarrow \searrow$ to them. (This projection represents the bottom element of our projection domain, and so is called the $\perp$ demand in the rest of this paper). This "demand on new variables" is the result demand of previous sections.

10.6 An operational approach

Strictness and absence analysis are just two of the analyses which GHC performs. Many of the others, such as "box demand" analysis, discover much more operational properties, such as whether or not a particular pointer need be passed to a function. Although Wadler and Hughes' projection-based theory applies beautifully to the analyses in this paper, we cannot expect to extend it to cover operational properties also. We are working on a trace-based framework for demands to handle such properties, which we hope will appear in a future paper.

11 Implementation

SLPJ: Measurements and results

12 Related work

SLPJ: John: you are going to write this, right?

References

A Box demand analysis

NOTE: **This entire section is probably wrong in serious ways. But it's fumbling towards something useful, I think.**

We have now completed our tour of the strictness analyser. There, the demand domains are standard (apart from call demands), and so constituted a familiar framework in which to develop our realisation of the analysis. We now turn our attention to box-demand analysis, where the situation is reversed: the analysis is identical in form to that for strictness analysis, but the domains are much less familiar.

The purpose of box-demand analysis is to guide the unboxing transformation described in §2. Let us assume, for example, that a function takes a pair as its argument and is strict in the pair. Should we pass both components, or one or the other, or neither, or should we pass the pair itself? These are the questions that box-demand analysis will answer.

Note that even if only the components are passed, the called function can always reconstruct the tuple. Such *reboxing* is sound (because there is no notion of tuple identity or destructive component update), but may be inefficient, especially in a loop.

A.1 Developing intuition

Before we can proceed further, we need to say what we mean by the term "using the box of a value". The idea is that if a function "uses the box" of its argument, then the argument should be passed in boxed form to the function. Consider the following functions:

```
      -- Which of these "uses the box" for x?
f1 x = x                              -- Yes
f2 x = (x,True)                       -- Yes
f3 x = case x of (a,b) -> (b,a)       -- No
f4 x = case x of (a,b) -> (x,a)       -- Yes
f5 x = f3 x + 1                       -- No

f6 x = (True, case x of (a,b) -> a)   -- Yes

f7 x y = if (case x of (a,b) -> a)    -- No
         then
            (True,  case x of (a,b) -> a)
         else
            (False, y)
```

Here, `f1` uses the box for `x`, because it returns `x` itself; `f2` uses the box for `x` because it builds `x` into a pair. On the other hand, `f3` does not use the box for `x` *because the only use it makes of* `x` *is to take it apart*; the worker for `f3` should be passed only the components of `x`, not `x` itself. The function `f4` takes `x` apart, like `f3`, but it also builds `x` into a pair (like `f2`) so it *does* use the box. If the box is not passed to the worker for `f4`, the worker will have to re-box the pair.

The next function, `f5`, illustrates the transitive property we seek: If `x` is passed to a function that itself does not use the box, then we want that property to propagate to the caller; in this case, `f5` does not use the box because `f3` does not.

Box-demand and strictness are not altogether separable. For example, `f5` does use the box for `x`, because it is lazy in `x`, so `x` must be passed unevaluated (and hence boxed) to the function. However, `f7` illustrates a subtle point that we missed altogether in the beginning. It is certainly strict, and certainly does not use the box for `x`. But if we consider only the `then` branch, we see it is precisely the same as `f6`'s right hand side. So it would be wrong to say that the `then` branch, being lazy in `x`, uses the box for `x`, because there may be other reasons why the function in which the sub-expression occurs is strict in `x`.

The conclusion is this: the box-demand analysis must treat *sub-expressions* and *functions* differently. For example, the expression

```
(True, case x of (a,b) -> a)
```

does not use the box for `x`, but the function

```
(\x -> (True, case x of (a,b) -> a))
```

does use the box for its argument. We will elaborate this point in §A.4.

A.2 Dealing with conditionals

Consider the following expression:

```
f x as = case as of
           []     -> x
           (b:bs) -> case x of (a,b) -> a
```

Does `f` "use the box" of `x` or not? In one of the `case` branches, the answer is 'Yes', while on the other the answer is 'No'. So what are we to say for the whole expression? Our choice is to say that `f` does *not* use the box of `x`. Here is the example that convinced us:

```
last2 :: (Int,Int) -> [Int] -> (Int,Int)
last2 p []     = p
last2 p (x:xs) = case p of (a,b) -> last2 (b,x) xs
```

In the base case, `last2` returns the pair `p`. In the recursive case, it builds a new pair before recursing. Should we pass the boxed pair to `last2` on the grounds that one branch uses the box? By no means! On every iteration except the last, we would simply take apart the newly constructed pair, so we can save a great deal of allocation by instead accepting the reboxing cost in the base case.

Unfortunately, one can also construct examples where the reverse is true:

```
wug :: (Int,Int) -> [Int] -> [(Int,Int)]
wug p []     = case p of (a,b) -> [(b,a)]
wug p (x:xs) = case p of (a,b) -> if x==a
                                  then p : wug p xs
                                  else     wug p xs
```

Here, in the base case we take the pair apart, while in the recursive case we also use the box; and `wug` is still strict in `p`. If we pass `p` unboxed (i.e. just pass the components) we will incur a reboxing cost each time round the loop.

One can imagine many clever schemes, but we have adopted one simple one: we regard an expression as using the box only if every execution path (i.e. case branch) uses the box. In our system, neither `last2` nor `wug` use the box. In practice this seems to work pretty well. Accumulating-parameter functions like `last2` are common, whereas `wug` is highly contrived (hence its name).

John: These examples suggest that we should assign case branches weights, perhaps by profiling, perhaps by analysis, and say the box is used if it used in branches whose weight sums to at least 50%. Do we have evidence that this is not worthwhile? Perhaps we should mention the possibility, anyway.

$$
\begin{array}{rcll}
\mathcal{T}_b[\![\texttt{Int\#}]\!] & = & \{\bot, A, B, T\} & \text{Unlifted integers} \\
\mathcal{T}_b[\![\texttt{Int}]\!] & = & U(\mathcal{T}_b[\![\texttt{Int\#}]\!]) & \text{Lifted integers} \\
 & \cup & \{\bot, A, B\} & \\
 & = & \{\bot, A, B, U(\bot), U(A), U(B), U(T)\} & \\
\mathcal{T}_b[\![(t_1,t_2)]\!] & = & U(\mathcal{T}_b[\![t_1]\!], \mathcal{T}_b[\![t_2]\!]) & \text{Product types} \\
 & \cup & \{\bot, A, B\} & \\
\mathcal{T}_b[\![\texttt{[t]}]\!] & = & \{\bot, A, B, T\} & \text{Sum types} \\
\mathcal{T}_b[\![t_1\texttt{->}t_2]\!] & = & S(\mathcal{T}_b[\![t_2]\!]) & \text{Functions} \\
 & \cup & \{\bot, A, B\} & \\
\mathcal{T}_b[\![\alpha]\!] & = & \{\bot, A, B, T\} & \text{Unknown types}
\end{array}
$$

Figure 13: Box-demands at various types

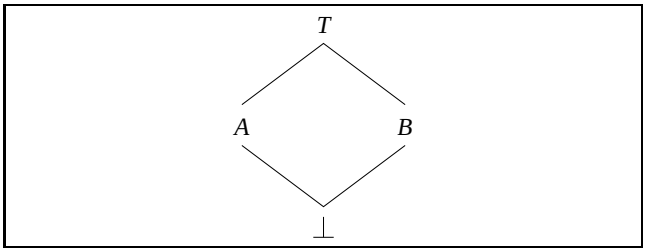

Figure 14: Box demands for unlifted integers (`Int#`)

A.3 The box-demand domain and its operations

Based on these intuitions, we are now ready to present the domain of box demands. The syntax of box demands is

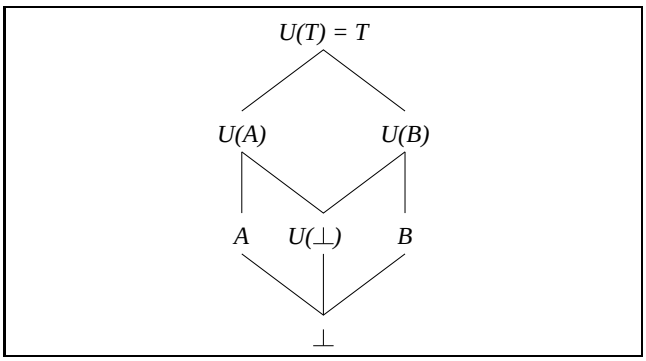

Figure 15: Box demands for lifted integers (`Int`)

The set *BDemand* is the set of box demands b defined below.

Box demand	b	::=	$\bot$	Needed in a divergent computation
		|	A	Absent, definitely not used
		|	B	Box definitely used
		|	$U(\overline{b})$	Box may be unused, components $\overline{b}$
		|	$S(b)$	Call, with result box demand b
		|	T	No information
Box demand sequence:	$\overline{b}$	::=	$[b_1, \ldots, b_n]$	Tuple components
		|	$\hat{b}$	Length-polymorphic tuple components

Figure 12: Box demands

shown in Figure 12. As in the case of strictness analysis, the actual domain depends on the data type (Figure 13). However, every box-demand domain contains the following four basic elements, arranged in a diamond lattice, as shown in Figure 14. An expression e may place the following box-demands on a variable `x`:

- A (Absent): `x` is not mentioned anywhere in e; the wrapper need not pass the argument to the worker at all.
- B (Box): every execution path in e uses the box of `x`; the wrapper should pass the box to the worker.
- T (Top): not all execution paths use the box for `x`; if the function is strict, the wrapper should pass the components, but not the box, to the worker.
- $\bot$ (Bottom): e goes into an infinite loop, or calls the `error` function.

The discussion about conditionals, in the previous section, explains why B contains definite information ("every path...") whereas T is less defined ("not every path...").

These four points suffice for unlifted types, sum types, and unknown types (Figure 13), but we need more structure for product and function types:

- $U(\overline{b})$ (Unboxed): used for product types only. Not all execution paths need the box (the product itself), and the joint need for each component is described by $\overline{b}$. If the function is strict, the wrapper may pass the components described by $\overline{b}$ instead of the box.

  We identify $U(\hat{T})$ with T, much as we identify $S(\hat{\bot}) = \bot$ in the strictness analyser.
- $S(b)$ (Call): a call demand, used for function types only. Call demands work in exactly the same way as for strictness analysis. The value is a function, which when applied will have box demand b imposed on its result. Like U, we identify $S(T) = T$.

The ordering on box demands is shown in Figure 16, together with the 'lub' and 'both' operators. *SLPJ: How can we justify these definitions? It's a big Figure to give without elaboration, but it's a bit hard to justify except by armwaving. Are any of the definitions surprising, worthy of more explanation?*

Notice that we *cannot* identify $U(A)$ with A. To see why, consider

```
a1, a2 :: Int -> [Bool]
a1 x = [ x `seq` True ]
a2 x = [True]
```

The function `a1` places box-demand $U(A)$ on `x`, whereas `a2` places demand A. It would be patently wrong to say that `x` is not used at all by `a1`.

Similar reasoning shows that we must distinguish $U(\bot)$ from $\bot$. Consider

```
a3 :: Int -> Bool -> [Bool]
a3 x b = if b then
              error (a1 x)
         else
              [b]
```

The demand on `x` arising from the call `error (a1 x)` is $\bot \,\&\, U(A)$ — the $\bot$ comes from the fact that `error` diverges, while the $U(A)$ comes from the call of `a1`. By Figure 16,

$$\bot \,\&\, U(A) \;=\; U(\bot \& A) \;=\; U(\bot)$$

Now if $U(\bot) = \bot$, the overall demand on `x` from `a3` would be $A \sqcup \bot = A$, which is plainly wrong.

We could have additionally permitted structured box demands $B(\overline{b})$, meaning that all branches need the box and in addition the components are needed $\overline{b}$ much. But we chose to use simple B only, on the grounds that if we pass the box, then the runtime advantage of additionally passing some components is dubious.

A.4 The box demand analysis function

The box demand analysis function, $\mathcal{B}[\![e]\!]$, has almost exactly the same form as the strictness analyser, $\mathcal{S}[\![e]\!]$ (Figure 6), except that we use the box-demand domain in Figure 12, and its operators in Figure 16, instead of the strictness-demand domain and its operators. Since it is simply a stepping stone to the joint-demand analyser, which we present in detail later, we will briefly summarise the differences between $\mathcal{S}[\![e]\!]$ and $\mathcal{B}[\![e]\!]$, rather than giving the full definition of the latter.

There are several important differences. Firstly, the "vanilla demand" (used when approximating a demand transformer) is B, not S; that is, the vanilla demand uses the box of the returned value.

$$
\begin{array}{rcl}
\bot & \sqsubseteq & b \\
A & \sqsubseteq & U(\overline{A}) \\
B & \sqsubseteq & U(\overline{B}) \\
\overline{b_1} \sqsubseteq \overline{b_2} \;\Rightarrow\; U(\overline{b_1}) & \sqsubseteq & U(\overline{b_2}) \\
b & \sqsubseteq & T \\
b_1 \sqsubseteq b_2 \;\Rightarrow\; S(b_1) & \sqsubseteq & S(b_2)
\end{array}
$$

$$
\begin{array}{rcll}
\bot & \sqcup & b & = b \\
\\
A & \sqcup & A & = A \\
A & \sqcup & B & = T \\
A & \sqcup & S(b) & = T \\
A & \sqcup & U(\overline{b}) & = U(A \sqcup \overline{b}) \\
\\
B & \sqcup & B & = B \\
B & \sqcup & U(\overline{b}) & = U(B \sqcup \overline{b}) \\
B & \sqcup & S(b) & = S(b) \\
\\
U(\overline{b_1}) & \sqcup & U(\overline{b_2}) & = U(\overline{b_1} \sqcup \overline{b_2}) \\
U & \sqcup & S(b) & = T \\
\\
S(b_1) & \sqcup & S(b_2) & = S(b_1 \sqcup b_2) \\
\\
T & \sqcup & b & = T \\
{\scriptstyle a} & & &
\end{array}
$$

$$
\begin{array}{rcll}
\bot & \& & \bot & = \bot \\
\bot & \& & A & = \bot \\
\bot & \& & B & = B \\
\bot & \& & S(b) & = S(\bot \& b) \\
\bot & \& & U(\overline{b}) & = U(\bot \& \overline{b}) \\
\bot & \& & T & = B \\
\\
A & \& & b & = b \\
\\
B & \& & B & = B \\
B & \& & S(b) & = S(b) \\
B & \& & U(\overline{b}) & = U(\overline{B \& b}) \\
B & \& & T & = B \\
\\
S(b_1) & \& & S(b_2) & = S(b_1 \& b_2) \\
S(b_1) & \& & U(\overline{b}) & = S(b_1) \\
S(b) & \& & T & = S(b) \\
\\
U(\overline{b_1}) & \& & U(\overline{b_2}) & = U(\overline{b_1} \& \overline{b_2}) \\
U(\overline{b}) & \& & T & = U(T \& \overline{b})
\end{array}
$$

[a] *John: These definitions are not consistent. $\sqcup$ should be the least upper bound operator for the $\sqsubseteq$ ordering, which demands for example that $B \sqsubseteq S(b)$. Call demands have been forgotten completely in the definition of $\sqsubseteq$.*

Figure 16: Ordering and operations on box demands

Secondly, the lookup function defined in Figure 5 returns A, not L, for variables not in the domain of θ:

$$
\begin{array}{lcll}
\theta(\mathtt{T})x & = & A & \text{if } x \notin dom(\theta) \\
\theta(\mathtt{X})x & = & \bot & \text{if } x \notin dom(\theta) \\
\theta(r)x & = & \theta(r) & \text{if } x \in dom(\theta)
\end{array}
$$

In the same way, the first equation of the box-demand analyser looks like this:

$$\mathcal{B}[\![e]\!]\ \rho\ A\ =\ \mathtt{T} \Rightarrow [\,][\,]$$

(see the first equation of Figure 6). That is, if we analyse e with an absent demand, we return a demand type that places only an absent demand on all its free variables.

Thirdly, in §A.1 we showed that the box-demand analysis must treat sub-expressions and functions differently. This difference can be neatly catered for in the equation for a called lambda:

$$
\begin{array}{rcll}
\mathcal{B}[\![\backslash x.e]\!]\ \rho\ S(s) & = & \textit{let}\ \ r \Rightarrow \sigma\theta & = \mathcal{S}[\![e]\!]\ \rho\ s \\
 & & \qquad\quad s_x & = \theta(r)x \\
 & & \textit{in}\ r \Rightarrow \mathcal{F}(s_x) : \sigma\theta\backslash\{x\} &
\end{array}
$$

(see the corresponding equation in Figure 6). When the demand s_x becomes a function argument, we must force it to be boxed unless the function is strict, using the auxiliary function $\mathcal{F}(s_x)$:

$$
\begin{array}{rcll}
\mathcal{F}(A) & = & A & \\
\mathcal{F}(U(\overline{s})) & = & U(\overline{\mathcal{F}(s)}) & \\
\mathcal{F}(s_x) & = & s_x & \text{if demand on } x \text{ is strict} \\
 & = & B & \text{otherwise}
\end{array}
$$

(This definition of $\mathcal{F}(s_x)$ informally assumes that we have strictness information available, which we will have when we combine the two analyses in the next Section.) Notice that $\mathcal{F}(\cdot)$ must be applied recursively to the components of a U demand. *John: What?? This seems to say that if the demand on the argument is unboxed ($U(s)$), then the demand type says the argument is unboxed regardless of whether the function is strict or not! I'm assuming that the second equation "takes precedence" over the third, since it is more specific. This is a mixup, surely?*

These are the major differences between the two analysers. There are other minor ones — for example, the treatment of $\mathcal{B}[\![\backslash x.e]\!]\ \rho\ U(\overline{b})$ — but rather than present them in detail, we proceed to the joint-demand analyser.

A.5 The `error` function again

The box-demand analyser also shows that we really do need to keep a free-variable map of a demand type even when the result-type component is `X`, something that is not necessary for strictness (§5.5). Consider the following box-demand analysis:

$$\mathcal{B}[\![\texttt{error x}]\!]\ \rho\ B\ =\ (\mathtt{X}, [\mathtt{x} \mapsto B])$$

The `error` function has box-demand signature `B -> X`; that is, it uses the box of its argument, and diverges (`X`). It is important to remember that `x` is used, via the mapping, otherwise it will be taken as absent, which is not right (c.f. §2.2).

B Joint demand analysis

We have now described two separate backwards analysers, one for strictness and one for box-demand. Since their form is identical, it is natural to ask whether we can compute both analyses at once. The way to do so is to combine their domains together.

One way to do so would be to take the cartesian product of the two domains, but in practice the two components will often have the same structure, so the product space would have more points than necessary. Hence, we define a single "zipped up" domain, which captures the essential information, and describe its meaning by injecting it into the cartesian product domain.

B.1 Zipped-up demand representation

SLPJ: I am concerned about how to motivate the choice of elements of this domain, and its type translation. For example, should Err be in the `Int#` *domain; why do we eliminate one out of the four points we'd get from taking the cartesian product? Should Err be in the* `->` *domain?*

SLPJ: There's also a difficulty with having Seq in the α domain, because its semantics is $\langle S(\hat{L}), U(\hat{A})\rangle$, but $U(\hat{A})$ isn't in the box-deman domain for α. Maybe it should be? But then we have five points, not four. Maybe we abbreviate $U(\hat{A})$ as plain U?

Figure 17 shows the syntax of joint demands. We explain what each of these joint demands means using the function $\mathcal{I}[\![]\!]$, which injects each joint demand into the product space $StrDmd \times BDemand$. We can give intuitions for each demand, as follows:

- *Bot* means that the argument will be used only in non-terminating expressions; its semantics is simply $\langle \bot, \bot \rangle$.
- *Abs* means that the argument will not be evaluated nor used at all. Its semantics is $\langle L, A \rangle$: the strictness component says that it is lazy, while the box-demand component says that it is not used.
- *Top* means that we know nothing about the use of the argument.
- $Call(d)$ means that the argument is a function, that the value is needed, and when the function is applied (to one argument), the demand on its result is d. Its semantics is simply a combination of the call demands from the strictness and box-demand lattices.

  Comment: I've changed if to when above — a call demand Call(d) does mean that the function will be applied, doesn't it?
- $Eval(\overline{d})$ means that the argument is used strictly, and its components are demanded as described by $\overline{d}$. Its semantics are given by a combination of S on the strictness side, and U on the box-demand side.
- The demand $Defer(\overline{d})$ means that the argument may or may not be evaluated, but that the box demand on it is described by the box-demand component of d. We noted in §A.1 the importance of recording box-demand information even in lazy contexts.

$$
\begin{array}{rcl}
d & ::= & Bot \\
 & | & Top \\
 & | & Abs \\
 & | & Call(d) \\
 & | & Eval(\overline{d}) \\
 & | & Defer(\overline{d}) \\
 & | & Box(d) \\
 & & \\
\overline{d} & ::= & d_1 \ldots d_n \\
 & | & \hat{d}
\end{array}
$$

Semantics

$$
\begin{array}{lcl}
\mathcal{I}[\![Bot]\!] & = & \langle \bot, \bot \rangle \\
\mathcal{I}[\![Abs]\!] & = & \langle L, A \rangle \\
\mathcal{I}[\![Call(d)]\!] & = & \langle S(s), S(b) \rangle \\
\mathcal{I}[\![Top]\!] & = & \langle L, T \rangle \\
\mathcal{I}[\![Defer(\overline{d})]\!] & = & \langle L, U(\overline{b}) \rangle \\
\mathcal{I}[\![Box(d)]\!] & = & \langle s, B \rangle \\
\mathcal{I}[\![Eval(\overline{d})]\!] & = & \langle S(\overline{s}), U(\overline{b}) \rangle \\
 & & \text{where } \langle s, b \rangle = \mathcal{I}[\![d]\!]
\end{array}
$$

Abbreviations

$$
\begin{array}{rcl}
Lazy & = & Box(Top) \\
Err & = & Box(Bot) \\
Str & = & Box(Seq) \\
Seq & = & Eval(\widehat{Abs})
\end{array}
$$

Identities

$$
\begin{array}{rcl}
Defer(\widehat{Top}) & = & Top \\
 & & \\
Box(Abs) & = & Lazy \\
Box(Top) & = & Lazy \\
Box(Defer(\overline{d})) & = & Lazy \\
Box(Call(d)) & = & Call(d) \\
Box(Box(d)) & = & Box(d)
\end{array}
$$

Figure 17: Syntax and semantics of joint demands

- The demand $Box(d)$ means that the box is needed in every branch, with the strictness specified by the evaluation demand component of d.

Figure 17 also defines a few useful abbreviations:

- The demand *Lazy* means that the argument may or may not be needed.
- The demand *Seq* means that the expression will be evaluated, but its value will be ignored. This is the demand that `seq` imposes on its first argument.
- The demand $Err = \langle \bot, B \rangle$ is the demand that (say) `error s` imposes on `s`. It expresses the fact that the box of `s` is needed, but from a strictness point of view the demand on `s` is hyperstrict ($\bot$).

$$
\begin{array}{rcll}
\mathcal{T}_z[\![\texttt{Int\#}]\!] & = & \{\bot, Top, Abs\} & \text{Unlifted integers} \\
\mathcal{T}_z[\![(t_1\texttt{,}t_2)]\!] & = & Eval(\mathcal{T}_z[\![t_1]\!], \mathcal{T}_z[\![t_2]\!]) & \text{Products} \\
 & \cup & Box(Eval(\mathcal{T}_z[\![t_1]\!], \mathcal{T}_z[\![t_2]\!])) & \\
 & \cup & Defer(\mathcal{T}_z[\![t_1]\!], \mathcal{T}_z[\![t_2]\!]) & \\
 & \cup & \{\bot, Top, Abs, Err, Lazy\} & \\
\mathcal{T}_z[\![t_1\texttt{->}t_2]\!] & = & S(\mathcal{T}_z[\![t_2]\!]) & \text{Functions} \\
 & \cup & \{\bot, Top, Abs\} & \\
\mathcal{T}_z[\![\texttt{[t]}]\!] & = & \{\bot, Top, Abs, & \text{Sums} \\
 & & Err, Seq, Str, Lazy\} & \\
\mathcal{T}_z[\![\alpha]\!] & = & \{\bot, Top, Abs, & \text{Unknown types} \\
 & & Err, Seq, Str, Lazy\} &
\end{array}
$$

Figure 18: Type translation for zipped-up demands

- The demand $Str = Box(Eval(\widehat{Abs})) = \langle S(\hat{L}), B\rangle$ means that the argument will be evaluated, that its value is needed, and that its components may or may not be evaluated. This is the plain strict demand.

The lattice even for simple types, such as `Int` is now quite complicated, and intution begins to become an unreliable guide. For example, is it true that $Box(Box(d)) = Box(d)$? That is why we have presented the two analyses separately: we can now calculate the answer to such questions.

$$
\begin{array}{rcl}
Box(Box(d)) & = & Box(\langle s, B\rangle) \\
 & = & \langle s, B\rangle \\
 & = & Box(d) \\
 & & \text{where } \langle s, b\rangle = d
\end{array}
$$

(In this calculation we are slightly sloppy about writing calls to $\mathcal{I}[\![]\!]$, because they clutter up the reasoning.) Figure 17 gives some other useful identities that can be verified in the same way.

B.2 Operation over joint demands

Figure 19 gives the "lub" and "both" operators for joint demands. Again, we can *calculate* these functions, as the following example shows:

$$
\begin{array}{rcl}
Abs \sqcup Defer(d) & = & \langle L, A\rangle \sqcup \langle L, b\rangle \\
 & = & \langle L, A \sqcup b\rangle \\
 & = & Defer(Abs \sqcup d)
\end{array}
$$

We simply convert to the product space, perform the operation component-wise, and convert back (approximating if necessary).

B.3 The demand analysis function

The main analysis function, $\mathcal{J}[\![e]\!]$, in Figure 20, takes exactly the same form as the strictness and box-demand analysers already presented.

One difference is that the rather informal function $\mathcal{F}(\cdot)$, introduced in §A.4, can now be defined formally (Figure 21),

$$\langle s, b\rangle \in Demand = StrDmd \times BDemand$$

$$\langle s_1, b_1\rangle \sqsubseteq \langle s_2, b_2\rangle \quad \text{iff} \quad s_1 \sqsubseteq s_2 \wedge b_1 \sqsubseteq b_2$$

Peter to fill in here

Figure 19: Joint demand domain and its operations

$\mathcal{F}(d)$ takes a demand d for the free variable x of an expression e, and returns the demand appropriate for the argument of the function `\x->e`.
John: Why doesn't Eval have an argument in the fourth equation here?

$$
\begin{array}{rcl}
\mathcal{F}(Top) & = & Lazy \\
\mathcal{F}(Defer(\overline{d})) & = & Lazy \\
\mathcal{F}(Eval(\overline{d})) & = & Eval(\overline{\mathcal{F}(d)}) \\
\mathcal{F}(Box(Bot)) & = & Eval \\
\mathcal{F}(Box(d)) & = & Box(\mathcal{F}(d)) \\
\mathcal{F}(Bot) & = & Abs \\
\mathcal{F}(d) & = & d
\end{array}
$$

$\mathcal{D}_T(dt)$ takes a demand type dt and returns a "deferred" demand type, by discarding all the strictness information from dt, retaining only box-demand information.
John: What does the third line mean here? It's a bit confusing to omit $\mathcal{I}[\![]\!]$ everywhere.

$$
\begin{array}{rcl}
\mathcal{D}_T(r \Rightarrow \sigma\theta) & = & \mathtt{T} \Rightarrow [\,]\mathcal{D}_\theta(\theta) \\
\mathcal{D}_\theta(\theta) & = & [x \mapsto \mathcal{D}(\theta(x)) \mid x \in dom(\theta)] \\
 & & \\
\mathcal{D}(\langle s, b\rangle) & \sqsupseteq & \langle L, b\rangle \\
 & & \\
\mathcal{D}(Bot) & = & Abs \\
\mathcal{D}(Abs) & = & Abs \\
\mathcal{D}(Top) & = & Top \\
\mathcal{D}(Call(d)) & = & Lazy \\
\mathcal{D}(Box(d)) & = & Lazy \\
\mathcal{D}(Defer(\overline{d})) & = & Defer(\overline{d}) \\
\mathcal{D}(Eval(\overline{d})) & = & Defer(\overline{d})
\end{array}
$$

Figure 21: Auxiliary functions for joint demand analysis

because the demand embodies strictness information as well as boxity.

The main difference between the joint analysis and the separate ones is described next.

B.4 Deferred demands

Consider what happens when we analyse an expression with a lazy demand, such as $Lazy$. The strictness analyser stopped at this point, returning the top demand type (first

$\mathcal{J}[\![e]\!]\ \rho\ d$ takes an expression e, a demand environment ρ and an evaluation demand d, and computes a demand type dt for e.

$$
\begin{array}{rcll}
\mathcal{J}[\![e]\!] : StrEnv \to JDemand \to DemandType & & & \\
\rho : StrEnv & = & Var \to DmdSig & \\
\\
\mathcal{J}[\![e]\!]\ \rho\ Abs & = & \mathtt{T} \Rightarrow [\,][\,] & \\
\\
\mathcal{J}[\![e]\!]\ \rho\ d & = & \mathcal{D}_T(\mathcal{J}[\![e]\!]\ \rho\ Str) & \text{where } d \text{ is non-strict} \\
\\
\mathcal{J}[\![P]\!]\ \rho\ d & = & \mathcal{PT}(P,\ d) & \text{where } P \text{ is a product constructor} \\
\\
\mathcal{J}[\![x]\!]\ \rho\ d & = & \multicolumn{2}{l}{\begin{array}{lcll} let\ r \Rightarrow \sigma\theta & = & \mathcal{DT}(\rho(x),\ d) & \text{if } x \in dom(\rho) \\ & = & \mathtt{T} \Rightarrow [\,][\,] & \text{otherwise} \\ in\ r \Rightarrow \sigma\theta\ \&\ [\mathtt{x} \mapsto d] & & & \end{array}} \\
\\
\mathcal{J}[\![e_1\ e_2]\!]\ \rho\ d & = & \multicolumn{2}{l}{\begin{array}{lcl} let\ \ r_1 \Rightarrow d_a : \sigma_1\theta_1 & = & \mathcal{J}[\![e_1]\!]\ \rho\ S(d) \\ \quad r_2 \Rightarrow \sigma_2\theta_2 & = & \mathcal{J}[\![e_2]\!]\ \rho\ d_a \\ in\ r_1 \Rightarrow \sigma_1\theta_1\ \&\ r_2 \Rightarrow \sigma_2\theta_2 & & \end{array}} \\
\\
\mathcal{J}[\![\backslash x.e]\!]\ \rho\ Call(d) & = & \multicolumn{2}{l}{\begin{array}{lcl} let\ \ r \Rightarrow \sigma\theta & = & \mathcal{J}[\![e]\!]\ \rho\ d \\ \quad d_x & = & \theta(r)x \\ in\ r \Rightarrow \mathcal{F}(d_x) : \sigma\theta \backslash \{x\} & & \end{array}} \\
\\
\mathcal{J}[\![\backslash x.e]\!]\ \rho\ d & = & \mathcal{D}_T(\mathcal{J}[\![\backslash x.e]\!]\ \rho\ Call(Str)) & \\
\\
\mathcal{J}[\![\mathtt{case}\ e\ \mathtt{of}\ (x,y)\ \mathtt{->}\ a]\!]\ \rho\ d & = & \multicolumn{2}{l}{\begin{array}{lcl} let\ \ r_a \Rightarrow \sigma_a\theta_a & = & \mathcal{J}[\![a]\!]\ \rho\ d \\ \quad r_e \Rightarrow [\,]\theta_e & = & \mathcal{J}[\![e]\!]\ \rho\ S(\theta_a(r_a)x, \theta_a(r_a)y) \\ in\ r_a \Rightarrow \sigma_a\theta_a \backslash \{x,y\}\ \&\ r_e \Rightarrow [\,]\theta_e & & \end{array}} \\
\\
\mathcal{J}[\![\mathtt{case}\ e\ \mathtt{of}\ \overline{p_i\ \mathtt{->}\ a_i}]\!]\ \rho\ d & = & \multicolumn{2}{l}{\begin{array}{lcl} let\ \ \overline{r_i \Rightarrow \sigma_i\theta_i} & = & \overline{\mathcal{J}[\![a_i]\!]\ \rho\ d} \\ \quad r_e \Rightarrow \sigma_e\theta_e & = & \mathcal{J}[\![e]\!]\ \rho\ S \\ in & & \\ (\bigsqcup \overline{r_i \Rightarrow \sigma_i\theta_i \backslash fv(p_i)})\ \&\ r_e \Rightarrow \sigma_e\theta_e & & \end{array}} \\
\\
\mathcal{J}[\![\mathtt{letrec}\ x\ \mathtt{=}\ e\ \mathtt{in}\ b]\!]\ \rho\ d & = & \multicolumn{2}{l}{\begin{array}{lcl} letrec\ \ n & = & arity(e) \\ \quad r \Rightarrow [d_1, \ldots, d_n]\theta & = & \mathcal{J}[\![e]\!]\ \rho'\ Call^n(Str) \\ \quad sig & = & d_1 \to \ldots \to d_n \to (r, \theta) \\ \quad \rho' & = & \rho[x \mapsto sig] \\ in\ \mathcal{J}[\![b]\!]\ \rho'\ d & & \end{array}}
\end{array}
$$

$\mathcal{PT}(P,\ d)$ takes a product constructor C and a demand d, returns the constructor's demand type given that demand.

$$
\begin{array}{rcll}
\mathcal{PT}(P,\ Call^m(Eval(d_1, \ldots, d_n))) & = & \mathtt{T} \Rightarrow [d_1, \ldots, d_n][\,] & \text{if } n = m = arity(P) \\
& = & \mathtt{T} \Rightarrow [\,][\,] & \text{otherwise}
\end{array}
$$

$\mathcal{DT}(sig,\ d)$ takes a demand signature sig and a demand d, and applies the demand transformer described by sig to d, returning a demand.

$$
\begin{array}{rcll}
\mathcal{DT}(d_1 \to \ldots \to d_n \to (r, \theta),\ Call^m(d)) & = & r \Rightarrow [d_1, \ldots, d_n]\theta & \text{if } n \leq m \\
& = & \mathcal{D}(\mathtt{T} \Rightarrow [\,]\theta) & \text{otherwise}
\end{array}
$$

Figure 20: The joint demand analysis

equation in Figure 6, but that is not right for the joint demand analyser, because the demand type it returns distinguishes variables that are absent from those that are mentioned. Only if the demand is *Abs* can we simply stop (first equation in Figure 20). On the other hand, we cannot simply proceed by recursing over the structure of the expression; consider

$$\mathcal{J}[\![\texttt{length xs}]\!]\ \rho\ Lazy$$

If we use the equation for applications, we will get a strict demand for `xs`, which is not right. The solution is given in the second equation of Figure 20:

$$\mathcal{J}[\![e]\!]\ \rho\ d\ =\ \mathcal{D}_T(\mathcal{J}[\![e]\!]\ \rho\ Str)\ \text{ where } d \text{ is non-strict}$$

If the incoming demand d is non-strict, we analyse the expression with a strict demand, and then apply the function $\mathcal{D}_T(dt)$ to the demand type thereby produced. $\mathcal{D}_T(\cdot)$, pronounced "defer", is defined in Figure 21. It throws away all the information about arguments and results, and applies $\mathcal{D}(\cdot)$ to the demand of each of the free variables mentioned in θ[3].

The function $\mathcal{D}(\cdot)$ is defined Figure 21, using the semantics of joint demands, by the equation:

$$\mathcal{D}(\langle s, b\rangle)\ \sqsupseteq\ \langle L, b\rangle$$

That is, $\mathcal{D}(\cdot)$ *discards the strictness information, but retains the boxity information. John: This isn't an equation, which baffled me! What we mean, I take it, is that* $\mathcal{D}(\langle s, b\rangle)$ *is the* least d *such that* $\mathcal{I}[\![d]\!]\ \sqsupseteq\ \langle L, b\rangle$.

Given this defining equation, together with the semantics of joint demands in Figure 17, the effect of $\mathcal{D}(\cdot)$ on each joint demand can simply be read off (Figure 21). The joint demands do not fully cover the product demand space; hence the approximation "$\sqsupseteq$" in defining equation for $\mathcal{D}(\cdot)$.

The importance of all this was illustrated by example `f7` in §A.1:

```
f7 x y = if (case x of (a,b) -> a)
         then
            (True,  case x of (a,b) -> a)
         else
            (False, y)
```

The second sub-expression `(case x of (a,b) -> a)` will be analysed with a lazy demand (because it is the argument of a lazy pair constructor). From a strictness point of view it is lazy in `x`, and from a boxity point of view it does not use `x`'s box. So we analyse the sub-expression with a strict demand, to give demand $Eval(Str, Abs)$ for `x`, and then apply $\mathcal{D}(\cdot)$ to that demand, to give $Defer(Str, Abs)$, which is $\langle L, U(BA)\rangle$. This is precisely the reason that *Defer* demands exist in the joint-demand lattice; approximating to *Lazy*, say, which is $\langle L, B\rangle$, would lose too much absence information.

Exactly the same situation arises when a lambda is analysed with a demand that is weaker than a call demand; we analyse with a vanilla call demand, and use $\mathcal{D}_T(\cdot)$ to discard the strictness information in its result.

[3] In principle we should also apply $\mathcal{D}(\cdot)$ to the argument demands, but in fact we lose no useful information by simply discarding the argument demands altogether, becusea their boxity information is in any case discarded by $\mathcal{F}(\cdot)$. *SLPJ: Do we need to say more? Less?*

Evaluation	Evaluation in the wrapper
$\bot$	Completely evaluate the argument.
$S(\overline{s})$	Evaluate argument to whnf; evaluate components according to $\overline{s}$.
$S(d)$	Evaluate the argument to whnf.
L	Do not evaluate argument.
Box	**Argument passing**
$\bot$	Pass the evaluated argument.
A	Do not pass the argument.
B	Pass the (possibly evaluated) argument.
$U(\overline{b})$	If argument is evaluated then pass components $\overline{b}$ much if available; else pass the (possibly evaluated) argument.
$S(b)$	Pass the (possibly evaluated) argument.
T	Pass the (possibly evaluated) argument.

Figure 22: What the wrapper does

B.5 The worker-wrapper split revisited

Figure 22 describes how the joint demands specified in Figure 17 drive the worker-wrapper split.

Note that since Str abbreviates $\langle S(\hat{L}), B\rangle$, an argument with demand Str will be evaluated and its value (but no components) will be passed to the worker.

Comment: Demonstrate the worker-wrapper split on a few examples.

John: Why pass the evaluated argument when the box demand is $\bot$*? Wouldn't is be just as valid not to pass the argument? (Monotonicity is worrying me).*

John: What does passing components "if available" mean? Surely if the argument is evaluated, then the components are available?

B.6 Summary

C Experience with the implementation

Comment: Increased or decreased precision, compiled program efficiency, compiler speed, ...

D What remains to be done

This is all very pragmatic and I would really like help with putting in place some formal underpinnings. In particular:

- I think it's fine to have a somewhat *ad hoc* lattice of demands, choosing points that are of pragmatic significance, and otherwise approximating like crazy. But there should be an underlying much more detailed lattice, based on some principles. For example, it seems that 'evaluate or not', 'use the box or not', and 'use the components or not' are independent properties. (See the dicussion about deferred demands in §B.4.)

  The consumer-properties currently shown in Figure 1 are a start in that direction.

- Along the same lines, I'd like to have a formal way to say how & is defined. At the moment I'm just sticking my finger in the air.

E Related work

TODO: Hughes 1988 and references therein. Check in particular whether any of Wray 1985 and 1986, Dybjer 1987, Hall 1987, Hughes 1985, Hughes 1987, Karlsson 1987 and Wadler 1987b discuss the handling of tuples. Burn's evaluation transformers? It is unlikely that any of these were used to drive a worker-wrapper split. Faxén 1996

Strictness analysis: Mycroft 1981, ...

About whether call-demands are new: *John: Sort of. I wrote a paper on higher-order backwards analysis 14 years ago (!), in which demands on functions were of the form* ⟨abstract argument, demand on result⟩. *It's not exactly the same, but it's close: the only difference is that I combined a forwards abstract interpretation to collect demand transformers, hence the abstract argument. Your call demands are a simplification of the same idea. The simplification is a good one, of course, but we should probably at least refer to my old paper.*

F Conclusion

Comment: Some words about the trickiness of designing an analysis that collects several kinds of information at the same time, whose results are used to guide a transformation with somewhat subtle effects, and which should obtain as much useful, approximate, information as fast as possible.