Theory and Practice of Demand Analysis in Haskell

ILYA SERGEY
IMDEA Software Institute, Spain
(e-mail: ilya.sergey@imdea.org)

SIMON PEYTON JONES
Microsoft Research, Cambridge, UK
(e-mail: simonpj@microsoft.com)

DIMITRIOS VYTINIOTIS
Microsoft Research, Cambridge, UK
(e-mail: dimitris@microsoft.com)

Abstract
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.

Contents
1 Introduction 2
2 Characterising the problem 3
  2.1 The worker-wrapper split in GHC 4
  2.2 seq and error 6
  2.3 Shortcomings of the existing analyser 7
3 Backwards analysis 7
4 Projections and Projection-Based Program Analyses 8
  4.1 The Core language and its semantics 8
  4.2 Projections 10
  4.3 Properties of projections 11
  4.4 Useful projections for $D_\infty$ 11
  4.5 Projection environments 12
  4.6 Projection types and projection transformers 13
  4.7 Safety of projection-based analysis 14
5 Absence Analysis 16
  5.1 Motivation 16
  5.2 A definition of the analysis 17
6 Strictness Analysis 17
  6.1 Extending the domain for strictness 19
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:

- 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 an elegant formulation of backwards analysis for a higher-order language, based on higher-order projections in Section 4. These higher-order projections, are new, and constitute our most substantial technical contribution. They lead directly to a rather compact, compositional implementation using call demands (Section ??).
- 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 cartesian product domain. Previously, 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 (Baker-Finch et al., 2004), 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 Section 8. 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.
- We describe and motivate a range of theoretical and engineering design choices. For example, we have found that it is very important to “look inside” products (§ 4.4); that nested definitions are very common and must be handled well (§ 4.5); and that simple approximations to the full demand transformer for a function work well in practice (§ 4.6).
- 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 (§??).
- We use a clever folk-lore technique to improve the speed of convergence, and give measurements of its effectiveness (§??).

The focus of the paper is twofold. 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 and prove soundness of the analysis with respect to a standard denotation semantics (Ilya: Actually, this is nove wrt. to the old draft.)

2 Characterising the problem

The default parameter-passing mechanism in Haskell is call-by-need, in which an argument must be passed as a heap-allocated thunk, or suspension, 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 (Mycroft, 1980; ?; ?; ?).
- 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 (?; Henglein & Jørgensen, 1994).

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 analysis than to exploitation, yet one can only understand what information we need from the demand analysis by understanding the use to which that information is put. So we focus initially on exploitation, 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:

\[ f :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \]

\[ f \ p = \langle \text{rhs} \rangle \]

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:

\[ f :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \]

\[ f \ p = \text{case } p \ \text{of} \]

\[ (a, b) \rightarrow \text{wf } a \ b \]

\[ ^1 \text{A real compiler would avoid splitting very small functions, such as } f \ \text{above, since they can be inlined bodily, which is better than splitting. For presentational purposes we use small examples regardless of this; you can always make them bigger!} \]
Theory and Practice of Demand Analysis in Haskell

\[ wf :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ wf \ a \ b = \text{let } p = (a, b) \text{ in } <\text{rhs}> \]

Now the wrapper, \( f \), can be inlined at every call site, so that the caller evaluates \( p \), passing only the components to the worker \( w \), 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 \( w \). Hence the need for absence analysis. Suppose, then, that we know that \( b \) is not needed. Then we can transform to:

\[ f :: (\text{Int}, \text{Int}) \to \text{Int} \]
\[ f \ p = \text{case } p \text{ of } (a, b) \to w \ a \]
\[ w :: \text{Int} \to \text{Int} \]
\[ w \ a = \text{let } p = (a, \text{error } "\text{abs}") \text{ in } <\text{rhs}> \]

Since \( b \) is not needed, we can avoid passing it from the wrapper to the worker; while in the worker, we can use \( \text{error } "\text{abs}" \) 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 <\text{rhs}> is strict in \( p \) it must presumably take it apart.** So inside <\text{rhs}> we may see “\text{case } p \text{ of ...}”. Since \( p \) is explicitly bound to a pair in \( w \), we can eliminate the \text{case} in <\text{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:

\[ f :: (\text{Int}, \text{Int}) \to \text{Int} \]
\[ f \ p = (\text{case } p \text{ of } (a, b) \to a) + 1 \]

Then the worker-wrapper transformation will produce:

\[ f :: (\text{Int}, \text{Int}) \to \text{Int} \]
\[ f \ p = (\text{case } p \text{ of } (a, b) \to w \ a) \]
\[ w :: \text{Int} \to \text{Int} \]
\[ w \ a = \text{let } p = (a, \text{error } "\text{Urk}") \text{ in } (\text{case } p \text{ of } (a, b) \to a) + 1 \]

Now, in the code for \( w \), we can inline the definition of \( p \) at its use in the \text{case}, simplify the case, and discard the now-dead binding for \( p \), giving:

\[ w :: \text{Int} \to \text{Int} \]
\[ w \ a = a + 1 \]

Does the reboxing binding still disappear if \( p \) is not scrutinised by an explicit \text{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 \( w \), and the \text{case} will cancel as before. Is **all** reboxing eliminated in this way? No, it is not, a problem that we discuss in \S 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 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. From a semantic point of view, error s should be considered identical to ⊥, or divergence. For example, consider this function:

\[
\begin{align*}
f [] & = \text{error} "urk" \\
f (x:xs) & = y
\end{align*}
\]

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 \( f \text{loop} \), where \text{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:

\[
\begin{align*}
g_1 x y & = g_1 y x \\
g_2 x y & = \text{error} x
\end{align*}
\]

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 this worker/wrapper split for \( g_2 \):

\[
\begin{align*}
g_2 x y & = \text{wg}_2 \\
\text{wg}_2 & = \text{let} x = \text{error} "\text{abs}" \\
& \quad \text{in} \text{error} x
\end{align*}
\]

This is obviously wrong, because the call error \( x \) will attempt to print the string \( x \), but we have not passed \( x \) to \( \text{wg}_2 \)! In short, we must be careful not to assume that \( x \) is absent simply because it is consumed by a “divergent” computation: the error function diverges, but uses its argument.

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 subtle but pervasive effect. For example, eta reduction is not valid in general:

\[
g_3 a b \neq (\lambda x \rightarrow g_3 a b x)
\]

The former is ⊥, while the latter is not, and the two can be distinguished by seq. So far as strictness and absence analysis is concerned, is this function strict in \( x \)?

\(^2\)In GHC, error raises an exception, a nice generalisation of the Haskell 98 behaviour (?).
An explanation of the lambda expression:

\[ g_4 x = (\lambda y \to x) \texttt{seq True} \]

No, it is not: evaluating the lambda does not evaluate its body; indeed we can even treat \( g_2 \) as absent in \( x \), since the latter will never be used. In short, \texttt{seq} can evaluate a function without calling it, and our analysis must recognise this distinction. (A sound alternative would be to treat \texttt{seq} as lazy in its first argument, but that would be foolish because programmers often use \texttt{seq} precisely to make their functions strict.)

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!

### 2.3 Shortcomings of the existing analyser

In the past, GHC used an analyser based on the classic technique of abstract interpretation (Cousot & Cousot, 1977; ?) to derive strictness and absence information; this information in turn drives the generation of specialised calling conventions. The old analyser is described in our earlier papers (?; Peyton Jones, 1996).

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:

\[ f x y z = \langle \texttt{rhs} \rangle \]

the analyser figures out whether \( f \) is strict in \( x \), \( y \), and \( z \) by computing \( (f \perp \top \top) \), \( (f \top \perp \top) \), and \( (f \top \top \perp) \), where \( \top \) is the top-most abstract value, and \( \perp \) is the bottom-most. If \( f \) is recursive, it iterates the process using the newly-computed approximation to \( f \). The difficulty here comes when \( \langle \texttt{rhs} \rangle \) contains nested recursive definitions. Then to compute \( (f \perp \top \top) \), for example, we must compute the abstract values of the nested definitions, given these particular bindings: \( x = \perp \), \( y = \top \), 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 Backwards analysis

The previous 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. Absence analysis would, for example, distinguish $g_1$ and $g_2$ in Section 2. The compiler uses usage-demand information to decide which fragments of the argument to pass to the specialised version of the function.

Earlier work by Wadler and Hughes (1987) showed that using backward analysis accommodates both strictness and absence analysis, and handles (recursive) data structures as well. Backwards analysis answers the following question:

If an expression $e$ is consumed by a demand $d$, what demand is placed on $e$’s free variables?

A demand expresses the degree to which a value is evaluated. For example, a pair might not be evaluated at all (A), or be evaluated to head normal form (S), or its first or second components might be evaluated ($(S,A)$, or $(A,S)$), or both $(S,S)$. The parentheses give a suggestive textual notation for these demands, which we formalise later. Similarly a function might be evaluated to head normal form (S), or might be applied and its result evaluated ($C(S)$), and so on.

Seen in this light, an expression $e$ is a demand transformer, that transforms a demand on $e$ into demands on $e$’s free variables. For example the expression $(\text{fst } x + y)$ transforms the demand $S$ on the result into a demand $(A,S)$ on $x$, and $S$ on $y$.

Similarly a function

$$f x y = \langle rhs \rangle$$

is a demand transformer that transforms a demand on the result of a call to $f$ to demands on the arguments of that call.

Backwards analysis is called “backward” because it computes information about the inputs of an expression from information about its output.

Before we can make an analysis we need to formalise what exactly a “demand” might be, which we do in Section 4. Then we show how to apply this theory to build absence (Section 5) and strictness (Section 6) analyses in the context of higher-order functions. Finally, we show that the two can be combined into a single analysis that does the whole job in one blow. Ilya: Do we need to show this?

### 4 Projections and Projection-Based Program Analyses

In this section we follow Wadler and Hughes by formalising demands as projections (1987). The original analysis and its experimental implementation for a subset of Haskell (Kubiak et al., 1991) was able to handle only first-order language. The main new twist of this work is that we introduce higher order projections to deal with curried functions.

SLPJ: Did W&B handle only a first-order language? Ilya: Correct, added a line about this.

#### 4.1 The Core language and its semantics

SLPJ: Are we only handling product constructors? Then we’d at least have to add if so that we get the lub behaviour of the RHSs. Or else we can handle sums directly.
Theory and Practice of Demand Analysis in Haskell

Types
\[ \tau ::= P \tau \mid \tau \rightarrow \tau \mid a \mid \forall a. \tau \]

Expressions
\[ e ::= x \quad \text{Variables} \]
\[ e \; e \quad \text{Applications} \]
\[ \lambda x \rightarrow e \quad \text{Lambda-expressions} \]
\[ P \quad \text{Data constructors} \]
\[ \text{let } x = e \; \text{in } e \quad \text{Let-bindings} \]
\[ \text{letrec } x = e \; \text{in } e \quad \text{Letrec-bindings} \]
\[ \text{case } e \; \text{as } x \; \text{of } P \rightarrow e \quad \text{Case-expressions} \]

Constructor signatures
\[ \Sigma ::= \emptyset \mid \Sigma, P^\alpha : \forall \alpha. \tau \rightarrow P \tau \]

Fig. 1. Syntax of the Core language

\[
\begin{align*}
\text{Env} &= \text{Var} \Rightarrow \varepsilon \; D_w \\
[e] : \text{Env} \Rightarrow \varepsilon \; D_w
\end{align*}
\]

\[
\begin{align*}
[x]_\rho &= \rho(x) \\
[e_1 \; e_2]_\rho &= \text{app}([e_1]_\rho, [e_2]_\rho) \\
[P]_\rho &= \text{Fun}(\lambda d_1 \ldots \lambda d_n.(d_1, \ldots, d_n)) \\
[\lambda x \rightarrow e]_\rho &= \text{Fun}(\lambda d.[e]_\rho) \text{ where } \rho' = \rho[x \mapsto [e]_\rho] \\
[\text{let } x = e_1 \; \text{in } e_2]_\rho &= [e_1]_\rho' \text{ where } \rho' = \rho[x \mapsto [e_1]_\rho] \\
[\text{letrec } x = e_1 \; \text{in } e_2]_\rho &= [e_2]_{\text{E}(x, e, \rho)} \text{ where } \text{E}(x, e, \rho) = \lambda \rho'. \rho \sqcup \rho'[x \mapsto [e]_{\rho}] \\
[\text{case } e \; \text{as } x \; \text{of } P \rightarrow e]_\rho &= \begin{cases} [e]_{\rho'[\tau \rightarrow e]} & \text{if } [e]_\rho = P \tau \text{ and } P \text{ is a case-branch} \\ \bot & \text{otherwise} \end{cases}
\end{align*}
\]

Fig. 2. Denotational semantics of Core

Ilya: I guess, you mean that we don’t a type for sums? If so, I’m not sure whether we should focus on types at all. The semantics handle sums, indeed, as case expression has multiple branches.

Figure 1 presents the syntax of expressions of the Core language, which is entirely conventional. Types include datatypes, function types \( \tau \rightarrow \tau \) or type variables \( a \). Signatures \( \Sigma \) provide definitions of the datatype constructors \( P_\alpha \) coupled with their arities \( \alpha \).

SLPJ: Is “domain” identical to “cpo”? Let’s use one or the other consistently.

SLPJ: Do we really want to define the denotation of \( e \) as a continuous function from environments? There’s no need for this, and we don’t define \( \text{Env} \) as an element of \( D_w \).

Don’t we rather define a two-argument function from expression and environment to \( D_w \)?

We use denotational semantics to give meaning to Core, as shown in Figure 2. The denotation \([e]_\rho\) of an expression \( e \) in an environment \( \rho \) is an element of the domain \( D_w \).
Following Benton et al. (2009), $D_\infty$ is defined as the least fixed point of the equation

$$D_\infty = F D_\infty$$

where $F$ is a strict bi-functor $F$ on cpos as follows:

$$F(D^-, D^+) = (D^- \Rightarrow_c D^+) + \prod_{i \in \Sigma} D^+_i + \ldots + \prod_{i \in \Sigma} D^+_k + 1_{bad} \bot$$

Ilya: Say something about criteria for signature well-formedness.

The notation $D^+ \times D^+$ denotes a cartesian product, $\prod_n D$ abbreviates $n$-ary products (1 if $n = 0$). The notation $C \Rightarrow_c D$ stands for the cpo induced by the space of continuous functions from the cpo $C$ to the cpo $D$. We use $1_{bad}$ notation to simply denote a single-element cpo—the bad subscript is added for readability. The notation $D_\bot$ is lifting that instruments a domain with a new bottom element $\bot$.

Environments $p$ are just continuous functions $Var \Rightarrow_c D_\infty$ from syntactic variables to elements of the domain $D_\infty$. SLPJ: Really? I thought it was a partial function, and not in $D_\infty$ at all.

In Figure 2 we use the special “tag” $Fun$ to construct elements of the summand $D_\infty \Rightarrow_c D_\infty$ of the domain $D_\infty$. Similarly, we construct elements of other summands using product constructors $P^\alpha_i$ as tags for embeddings $D_\infty \times \ldots \times D_\infty \Rightarrow_c D_\infty$. The combinator $app$ refers to the continuous embedding $D_\infty \times D_\infty \Rightarrow_c D_\infty$ (Ilya: Define it formally!).

**Proposition 4.1.** There exists a solution to the domain-recursive equation induced by $F_\Sigma$.

### 4.2 Projections

Wadler and Blott’s key insight was that we can use projections to model our informal notion of a “demand” or “degree of evaluation”.

**Definition 4.1.** A continuous function $p \in D \Rightarrow_c D$ is a projection (Scott, 1981) on a cpo $D$ if for every $d \in D$,

$$p \dvdash d \quad p \circ p = p$$

In words, projections are idempotent and only remove information from an object, so one can reformulate these properties in a point-free as follows (Wadler & Hughes, 1987):

$$p \subseteq ID = \lambda x.x$$

Projections model demands in the following way: a demand that evaluates certain parts of a data structure (and not others) can be modeled by a projection that does not touch the evaluated parts of the data structure, but smashes the un-evaluated parts to $\bot$. So, for example, the function $fst$ places a demand on its argument that evaluates the first
component of a pair, and ignores the second. This demand can be modeled by the projection

\[ p = \lambda (a, b). (a, \bot) \]

Now we can formalise our claim that this is indeed the demand placed by \( \text{fst} \), by showing that

\[ \text{fst} = \text{fst} \circ p \]

That is, you can apply \( p \) to \( \text{fst} \)'s argument to throw information away, and \( \text{fst} \) will not notice.

4.3 Properties of projections

We will refer to a set of all projections on a domain \( D \) as \( \text{Pr}(D) \). Projections possess a number of useful properties. For instance, they form a complete lattice under the point-wise \( \sqsubseteq \) ordering, with \( \top = \text{ID} \) and \( \bot = \lambda x \rightarrow \bot_D \). The least upper bound is defined point-wise on projections, so the following proposition holds:

**Proposition 4.2** (Wadler and Hughes (1987)). If \( P \) is a set of projections then \( \sqcup P \) exists and is a projection.

**Lemma 4.1.** Let \( p_1 \) and \( p_2 \) be projections. Then \( p_1 \sqsubseteq p_2 \implies p_1 \circ p_2 = p_1 \).

**Proof.** By the definition of projections, \( p_1 = p_1 \circ p_1 \sqsubseteq p_1 \circ p_2 \), since \( p_1 \sqsubseteq p_2 \). Conversely, since \( p_2 \sqsubseteq \text{ID} \), we have \( p_1 \circ p_2 \sqsubseteq p_1 \circ \text{ID} = p_1 \).

Wadler and Hughes employ the projections to describe an analysis that, given a projection \( p \) and an element \( d \in D \) will infer the least possible projection \( q \), such that the following safety condition is satisfied:

\[ p \ d = p (q \ d) \]

That is, if an element \( d \) is used in the context of \( p \) (i.e., only the result of application of \( p \) to \( d \) is considered), than the projection \( q \) describes how much information can be "removed" from \( d \) so the result would remain the same. Given the idempotence of projections, one can think of \( p \) as the best possible candidate for the role of \( q \). However, we can do better for some interesting cases, e.g., for \( d \) being a function. We will elaborate on this in Section 4.4.

Finding the best possible (i.e., the least) projection \( q \) for given \( p \) and \( d \) is the goal of a projection-based analysis. The following context strengthening lemma establish the connection between results of the analysis with respect to projection ordering:

**Lemma 4.2** (Context strengthening). If \( p_2 \ d = p_2 (q \ d) \) and \( p_1 \sqsubseteq p_2 \) then \( p_1 \ d = p_1 (q \ d) \).

**Proof.** Straightforward from Lemma 4.1: \( p_1 \ d = p_1 (p_2 \ d) = p_1 (p_2 (q \ d)) = p_1 (q \ d) \).

4.4 Useful projections for \( D_\infty \)

In our development we will be considering projections over the semantic domain \( D_\infty \). Four particular projections will be particularly useful:

- \( \text{ID} = \lambda x. x \), the identity projection. This corresponds to a demand that uses all of its argument.
• \(BOT = \lambda x. \bot\), the bottom projection, corresponding to a demand that ignores its argument.
• \((p, q)\) a product projection, corresponding to a demand that uses a pair with its components demands described by \(p\) and \(q\).
• \(p \to q\), a higher-order projection, to be discussed shortly.

Here are the definitions of pair and higher-order projections:

**Definition 4.2** (Higher-order projections). Let \(p, q\) be projections, \(f \in D_{\infty}\). Then

\[
(p, q)f = \begin{cases} 
(p d_1, q d_2) & \text{if } f \in (D_{\infty} \times D_{\infty}) \text{ and } f = (d_1, d_2) \\
\bot & \text{otherwise}
\end{cases}
\]

\[
(p \to q)f = \begin{cases} 
q \circ f \circ p & \text{if } f \in (D_{\infty} \Rightarrow_c D_{\infty}) \\
\bot & \text{otherwise}
\end{cases}
\]

As usual, we assume the operator \((\cdot \to \cdot)\) to be right-associative. The following proposition ensures that defined above operators indeed yield projections.

**Proposition 4.3.** Let \(p, q \in Pr(D_{\infty})\). Then \((p \to q) \in Pr(D_{\infty})\) and \((p, q) \in Pr(D_{\infty})\).

**Proof.** For the \((p \to q)\) case, let us take some \(f \in (D_{\infty} \Rightarrow_c D_{\infty})\), otherwise the proof is trivial as \((p \to q)f \equiv \bot\). That is, \(f\) is a continuous (and, hence, monotonic) function on \(D_{\infty}\). Since \(q\) is a projection, we have \(q \subseteq ID\), therefore \(f \circ q \subseteq f\) (by monotonicity of \(f\)). Similarly, \(p \subseteq ID\) implies \(\forall g, p \circ g \subseteq g\). Taking \(g = f \circ q\), we obtain \(p \circ f \circ q \subseteq f \circ q \subseteq f\), i.e., \((p \to q) \subseteq ID\). Idempotence of \((p \to q)\) follows from idempotence of \(p\) and \(q\).

For the \((p, q)\) case, recall that the order on pairs in \(D_{\infty}\) is established component-wise, so for any \(g = (d_1, d_2)\), we have \((p, q)g = (p d_1, q d_2) \equiv (d_1, d_2) = g\).

In the remainder of the paper we will consider projections on \(D_{\infty}\) only, referring to them simply as to “projections”. We will be also employing projections of tuples of arbitrary arity rather than only pairs.

**Example 4.1.** A simple family of projections that allow one to detect the “absence” of components can be defined taking two basic projections: \(ID\) and \(BOT = \lambda x. \bot\), and deriving higher-order projections using \((\cdot \to \cdot)\) and \((\cdot, \cdot)\) operators.

For instance, let us consider a function \(f = \lambda x. \lambda y. x\). One can wonder, when \(f\) is called, which of its arguments may not be passed at all due to this absence. This question can be formalized via the following reified form of the safety condition ??: what is the least projection \(q\), such that \(d : p \Rightarrow q\), where \(p = ID \Rightarrow ID \Rightarrow ID\). The answer for this particular example is easy to find, taking \(q = ID \Rightarrow BOT \Rightarrow ID\).

### 4.5 Projection environments

Value environments as partial functions \(Var \Rightarrow D_{\infty}\) play a crucial role in defining a denotational semantics for the Core language. In order to reason about correctness of projection derivation procedures with respect to the denotational semantics, we introduce projection environments: projections on value environments.
An intentionally defined projection environment is a pair \( \theta = (\phi, r) \) where \( \phi \) a partial function from variables to projections and a \( r \) is default projection. An application \( @ \) of a projection environment \( \theta \) to a value environment \( \rho \) is defined point-wise as follows:

\[
(\theta @ \rho)(x) = \begin{cases} 
\phi(x)(\rho(x)) & \text{if } x \in \text{dom}(\phi) \\
r(\rho(x)) & \text{otherwise}
\end{cases}
\]

The least upper bound of two projection environments is computed as follows:

\[
\theta_1 \sqcup \theta_2 = \langle [x \mapsto \theta_1(x) \sqcup \theta_2(x) \mid x \in \text{dom}(\theta_1) \cup \text{dom}(\theta_2)], r_1 \sqcup r_2 \rangle
\]

where \( \theta_1 = (\phi_1, r_1) \) and \( \theta_2 = (\phi_2, r_2) \).

We refer to the projection environment of the form \( \langle \{\}, \bot \rangle \) as \( \theta_\bot \) and \( \langle \{\}, \top \rangle \) as \( \theta_\top \).

The next proposition follows naturally:

**Proposition 4.4.** Projection environments with the defined above application operation are projections on value environments (seen as a cpo of continuous functions of type \((\text{Var} \rightarrow \text{D}_\infty) \Rightarrow c(\text{Var} \rightarrow \text{D}_\infty))\).

We use the notation \( \theta \sqcup [x \mapsto p] \) for least upper bound of \( \theta \)'s \( \phi \) and a partial function \([x \mapsto p]\) (the lub exists because of Proposition 4.2). Finally, \( \theta \setminus \{x, y, \ldots\} \) denotes removing variables \( \{x, y, \ldots\} \) from the domain of \( \theta \)'s \( \phi \).

### 4.6 Projection types and projection transformers

In order to start building projection-based analyses as procedures to infer projections by expressions, we need to state a few more extra definitions.

**Definition 4.3 (Projection types).** A projection type \( \hat{\tau} \) is a pair \( \langle \theta, q \rangle \) for some projection \( q \) and a projection environment \( \theta \).

Intuitively, projection types describe results of a projection analysis for open expressions, i.e., those containing free variables, hence the environment \( \theta \). Projection types form a cpo, and \( \sqcup \) is established correspondingly component-wise.

**Definition 4.4 (Projection transformers).** Projection transformer \( \mathcal{T} \) is a continuous function mapping a projection to a projection type.

One can think of an open expression as of a projection transformer. One can imagine a transformer, corresponding to some \( d \) as a table mapping a “put” projection \( p \) to a resulting “safe” projection \( q \), such that \( d : p \Rightarrow q \).

**Example 4.2.** For the following function

\[
g :: (\text{Int, Int}) \rightarrow [a] \rightarrow (\text{Int, Bool})
\]

\[
g (a, b, c) = \text{case } a \text{ of } \\
0 \rightarrow \text{error } "urk" \\
_ \rightarrow \lambda y \rightarrow \text{case } b \text{ of } 0 \rightarrow (c, \text{null } y) \\
_ \rightarrow (c, \text{False})
\]

a part of the projection transformer table will look as follows (projection environment components are omitted, as \( g \) does not contain free variables):
In the first line of the table the projections put do not provide enough information about what is going to happen with arguments so any non-trivial safe projection could be computed. However, in the third and forth lines the put projection indicates that the appropriate components of the result tuple are going to be projected to $\bot$. This information makes it possible to compute interesting safe projections on the right-hand side.

Lemma 4.2 hints a nice way to approximate a projection transformer by an abstract transformer $\hat{T}$ by taking a finite sequence of input projections $p_1 \subseteq \ldots \subseteq p_n$, such that the value $\hat{T}(p_i) = \hat{\tau}_i = \hat{T}(p_i)$ is computed precisely, and for any $p'$ we have

\[
\begin{align*}
\hat{T}(p') &= \hat{\tau}_i & \text{if } p_{i-1} \subseteq p' \subseteq p_i \\
\hat{T}(p') &= \hat{\tau}_0 & \text{if } p' \not\subseteq p_0 \\
\hat{T}(p') &= \top & \text{otherwise}.
\end{align*}
\]

In practice, it makes sense to take $n = 1$, i.e., approximate a transformer $\mathcal{T}$ by a “step” function for a particular $p$, such that $\mathcal{T}(p) = \hat{\tau}$ and for any $p'$

\[
\begin{align*}
\mathcal{T}(p') &= \hat{\tau} & \text{if } p' \not\subseteq p \\
\mathcal{T}(p') &= \top & \text{otherwise}.
\end{align*}
\]

Ilya: Draw a simple rectangular picture illustrating the above.

The transformer environments we are going to use in the remainder of the paper will be binding variables with two-point abstract transformers of the form $(p, \hat{\tau})$, where $\hat{\tau}$ is a “threshold” projection and $\hat{\tau}$ is a projection type returned by the transformers for all $p' \not\subseteq p$. The “update” operation for a transformer environment $\hat{\rho}$, a variable $x$, a separator projection $p$ and a projection type $\hat{\tau}$ is denoted as $\hat{\rho}[x \mapsto (p, \hat{\tau})]$. 

### 4.7 Safety of projection-based analysis

At this point we can already picture a syntax-driven projection-based analysis for Core as a function $\mathcal{P}[]$ from expressions to projection transformers, such that if

\[\mathcal{P}[e] \, p = (\theta, q),\]

then

\[p \, [e]_p = p(q \, [e]_{\theta \cdot p}).\]

This implication is close to what we need as a definition of a safe analysis, however, it does not account to modularity. More precisely, we might have already precomputed
transformers for some of free variables from $\rho$, which we want our analysis to take into account. Therefore, we need to pass this information to the analysis $\mathcal{P}[\cdot]$ as an extra component, hence the following definition.

**Definition 4.5** (Transformer environments). A transformer environment $\hat{\rho}$ is a partial function from variables to abstract projection transformers.

The following definition establishes a connection between value environments and transformer environments. Since projection types form a lattice and projection transformers are continuous functions, we can define a partial order on transformer environments.

As it follows from Definition 4.5, the transformers from $\hat{\rho}$ return not only projections, but also projection environments (i.e., $\hat{\rho}(x)(p) = \langle \theta, q \rangle$). What does this environment part mean? In fact it says, that if $\hat{\rho}(x)(p) = \langle \theta, q \rangle$ then when a projection $p$ is put on a value corresponding to $x$, the projections at least as big as $\theta$ should be put on other free variables in the scope. The following example gives the essence of this phenomenon:

```haskell
let x = 42
in let y = x + y
    in (y, 1)
```

If the analysis proceeds compositionally and we are interested in demands, put on all variables, defined above some point, then it would be incorrect to analyse the expression $(y, 1)$ with a transformer environment $\hat{\rho} = \{ y \mapsto \langle \text{ID}, \langle \theta, \text{ID} \rangle \}$, as the projection environment $\theta$ does not indicate that $x$ should not be "bottomed". Therefore, a correct environment would be $\hat{\rho}' = \{ y \mapsto \langle \text{ID}, \langle \{ x \mapsto \text{ID} \}, \text{ID} \rangle \}$. I.e., if a projection $\text{ID}$ is put on $y$, it unleashes the projection at least as big as $\text{ID}$ to be put on $x$ as well.

It is clear that in order to get an adequate statement about safety of an analysis with respect to the denotational semantics, we need to relate an actual value environment $\rho$ with a transformer environment $\hat{\rho}$. However, it is problematic because of the $\theta$-components in the results of $\hat{\rho}$, which provide information about variables, whereas $\rho$ operates with pure values only. In order to resolve this issue, we introduce an extra level of indirection, “expanding” value environments to a “closure-converted” representation.

**Definition 4.6** (Expanded value environments). An expanded value environment $\sigma$ is an element of $\Sigma = \text{Var} \rightarrow (\text{Env} \times (\text{Env} \Rightarrow cD_{\infty}))$. The flattening operation $\sigma : \Sigma \rightarrow \text{Env}$ on expanded environments is defined point-wise as follows:

$$\sigma(x) = f(\rho) \text{ where } \langle \rho, f \rangle = \sigma(x)$$

We relate expanded environments with transformer environments by the following definition:

**Definition 4.7** (Related expanded and transformer environments). $\sigma \bowtie \hat{\rho}$ iff $\text{dom}(\hat{\rho}) \subseteq \text{dom}(\sigma)$ and for all projections $p$

$$p \ (f \ \rho) = p \ (q \ (f \ (\theta \ @ \ p))) \text{ if } x \in \text{dom}(\hat{\rho})$$

$$f \equiv \text{const} \text{ otherwise}$$

where $\langle \rho, f \rangle = \sigma(x)$ and $\langle \theta, q \rangle = \hat{\rho}(x)(p)$. 
In words, Definition 4.7 means that the transformer \( \hat{\rho} (x) \) for any projection \( p \) delivers a safe projection type \( \langle \theta, q \rangle \) for the correspondent pair \( \langle \rho, f \rangle = \sigma (x) \), otherwise, if \( x \in \text{dom} (\sigma) \setminus \text{dom}(\hat{\rho}) \), \( f \) is a constant function. The later condition means that if no information about \( x \) is provided by \( \hat{\rho} \), its value in \( \sigma \) does not depend on any variables.

A second attempt to define safety of a projection-based analysis Armed with a definition of projection environments, expanded environments and the relation between them, we can define a safe analysis as a function \( P \cdot K \cdot \) from expressions and transformer environments to projection transformers.

**Definition 4.8.** A projection-based analysis \( P \cdot K \cdot \) is safe with respect to the denotational semantics \( [\cdot] \), if for all \( \sigma, \hat{\rho} \), such that \( \sigma \bowtie \hat{\rho} \), and for all projections \( p \)

\[
\mathcal{P}[e] p = (\theta, q) \implies \mathcal{P}[q] \theta = \mathcal{P}[\theta \cdot \rho] \sigma,
\]

where \( \theta \triangledown \sigma = \lambda x. (\theta (x)) (f_x (\theta @ \rho_x)) \) for \( \langle \rho_x, f_x \rangle = \sigma (x) \).

It is noteworthy that the deep application \( (\theta \triangledown \sigma) \) for any \( x \) restricts not only arguments \( \rho_x \) of the appropriate expanded environment component \( \sigma (x) = \langle \rho_x, f_x \rangle \), but also the result of the application \( f_x (\theta @ \rho_x) \). This is how the knot is tied for free variables of \( e \). For instance, if \( x \notin \text{dom}(\hat{\rho}) \) then \( f_x \) is a constant function, and by taking \( (\theta \triangledown \sigma) (x) = \theta (x) (f_{x, \lambda}) \), we take a projected value, corresponding to a variable \( x \) in \( e \).

Ilya: should I say more words about this beast?

In the remainder of the paper we will refer to projections as to demands.

## 5 Absence Analysis

This section provides a motivation and gives a formal description for a projection-based absence analysis.

### 5.1 Motivation

The absence analysis answers the following question: “given an expression \( e \) and an environment \( \rho \), which components of \( e \) and \( \rho \) are unused when computing a value \( [e]_\rho \)?”. For example, in the definition:

\[
f \ x \ y = \text{if} \ x \equiv 0 \ \text{then} \ f (x - 1) \ y \ \text{else} \ x
\]

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, as in the following example:

\[
st \ p = \text{case} \ p \ \text{of} \ \{ (x, y) \rightarrow x \}
\]

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.

It is easy to define in terms of projections, given the background from Section 4.2. The \( \text{BOT} \) projection can be applied to a component of a value, which is never used. Application of the projection \( ID \), therefore, means that the value is going to be used,
without specifying any other details. However, working with pairs or functions, we can infer more information about absence. The table for Example 4.2 gives a good intuition about making use of elaborated demands for absence. The demand \( p = U \rightarrow U \rightarrow U \) for a function of type \( \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \) means that the function is going to be applied to both arguments and its result is going to be used. Having \( q = U \rightarrow A \rightarrow U \) as a safe demand put on this function before \( p \) would mean that the second argument can be safely ignored (i.e., mapped to \( \bot \) before the function is applied). The same intuition is applied to products of demands.

### 5.2 A definition of the analysis

The formal definition of the absence analysis is presented in Figure 3. Several cases are worth mentioning in its description. Since the constant absent demand \( A \) is “oblivious”, being put, it allows one not to compute the demands for constituents of the expression, so the result of the analysis for \( A \) is \( \langle \theta \bot, A \rangle \), i.e., one is free to omit any of the free variables in the demand environment (i.e., map them to \( \bot \)) and put the absent demand \( A \) on the expression itself.

It is notable, that in the case \( (A J) x K \hat{\rho} p \) when analyzing a variable \( x \), such that \( x \in \text{dom}(\hat{\rho}) \), but \( p \supseteq p' \), where \( p' \) is a threshold demand for a two-point transformer \( \hat{\rho}(x) \), the transformer \( \hat{\rho}(x) \) should return a safe demand type \( \langle \theta, q \rangle = \langle \theta \top, U \rangle \), which maps all free variables to \( U \), according to the definition of \( \theta \top \). However, in the case of absence analysis we can do better by taking \( \theta = \langle [y \mapsto U \mid y \in \text{FV}(e)], A \rangle \) instead of \( \theta \top \), where \( e \) is a right-hand side of a binding, defining \( x \). I.e., variables not participating in evaluation of \( x \) are still turned into bottoms, which does not affect the semantics, but is crucial for the analysis precision in other cases, when demand environments are combined.

The following theorem states the correctness of the analysis \( \mathcal{A}[] \) with respect to the denotational semantics.

**Theorem 5.1.** The projection-based absence analysis \( \mathcal{A}[] \) is safe with respect to the denotational semantics \( \llbracket \cdot \rrbracket \).

**Proof.** Taking the constant absence demand \( p = A \), the proof becomes trivial, as the analysis is represented then by the first rule only and taking \( q = A \) and \( \theta = \theta \bot \) is safe. For \( p \neq A \) the proof is by induction on the size of an analysed expression, since the analysis is defined compositionally. For the full proof, see Appendix A. \( \square \)

### 6 Strictness Analysis

The theory from Section 4 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.

\(^3\)In the remainder of this section we are going to refer to the \( \text{BOT} \) projection as \( A \) (for “absent”) and to \( \text{ID} \) as \( U \) (for “used”).
$\mathcal{A}[e]_{\hat{\rho}} p$ takes an expression $e$, a transformer environment $\hat{\rho}$ and a demand $p$ and computes a demand type $(\theta, q)$ for $e$.

\[
\mathcal{A}[e]_{\hat{\rho}} \lambda x \to e \quad = \quad \langle \theta, q \rangle
\]

\[
\mathcal{A}[e]_{\hat{\rho}} \lambda x \to e \quad = \quad \langle \theta, q \rangle = \begin{cases} \hat{\rho}(x)(p) & \text{if } x \in \text{dom}(\hat{\rho}) \\ \langle \theta, U \rangle & \text{otherwise} \end{cases}
\]

\[
\mathcal{A}[e_1, e_2]_{\hat{\rho}} p \quad = \quad \langle \theta_1 \sqcup \theta_2, q_1 \rangle
\]

\[
\mathcal{A}[\text{case } e \text{ as } z \text{ of } (x,y) \to a]_{\hat{\rho}} p \quad = \quad \langle \theta, q \rangle = \mathcal{A}[a]_{\hat{\rho}} p
\]

\[
\mathcal{A}[\text{case } e \text{ as } x \text{ of } P_1 \to e_1]_{\hat{\rho}} p \quad = \quad \langle \theta, q \rangle = \mathcal{A}[e_1]_{\hat{\rho}} p
\]

\[
\mathcal{A}[\text{let } x = e_1 \text{ in } e_2]_{\hat{\rho}} p \quad = \quad \langle \theta, q \rangle = \mathcal{A}[e_2]_{\hat{\rho}} p
\]

\[
\mathcal{A}[\text{letrec } x = e_1 \text{ in } e_2]_{\hat{\rho}} p \quad = \quad \langle \theta, q \rangle = \mathcal{A}[e_2]_{\hat{\rho}} p
\]

\[
\mathcal{A}[\text{letrec } x = e_1 \text{ in } e_2]_{\hat{\rho}} p \quad = \quad \langle \theta, q \rangle = \mathcal{A}[e_2]_{\hat{\rho}} p
\]

\[
\tilde{\mathcal{E}}(x, e, \hat{\rho}, p) = \lambda \hat{\rho}' \left(\langle \theta_2, q_2 \rangle = (\mathcal{A}[e]_{\hat{\rho}'}, p) \text{ in } \hat{\rho}' \sqcup \hat{\rho}'[x \to (p, \langle \theta_2 \setminus \{x\}, q_2 \rangle)]\right)
\]

$PT(P, p)$ takes a product constructor $P$ and a demand $p$ and returns an approximation of the constructor’s demand type given that demand.

\[
PT(P, U \to \ldots \to U) = \langle \theta, p_1 \to \ldots \to p_n \to U \rangle \quad \text{if } n = m = \text{arity}(P)
\]

\[
= \langle \theta, U \rangle \quad \text{otherwise}
\]

Fig. 3. Absence analysis
6.1 Extending the domain for strictness

Wadler and Hughes’ solution to mode strictness 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 \( \xi \) ("lightning bolt"), such that \( \xi \sqsubseteq \bot \) (Wadler & Hughes, 1987). It is important to realize 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 \( f \xi = \xi \). At every other point, \( f \) retains the usual semantics.

Now, in the extended domain we distinguish between the bottom element, corresponding to non-termination if evaluated (\( \bot \)), and non-termination itself \( \xi \). We can therefore model evaluating a value as a projection \( S \), such that

\[
\begin{align*}
S \xi &= \xi \\
S \bot &= \xi \\
S x &= x, \text{ otherwise}
\end{align*}
\]

I.e., evaluating \( \bot \) leads to divergence.

In order to deal with a new bottom \( \xi \), we replace the projection combinators (\( \cdot \rightarrow \cdot \)), (\( \cdot, \cdot \)) with new ones, (\( \cdot \rightarrow \cdot \)), (\( \cdot \rightarrow \cdot \)), evaluating their arguments, defined correspondingly:

\[
(p \rightarrow q) f = \begin{cases} \\
\lambda d. p(f(qd)) & \text{if } f \in (D_m \Rightarrow c D_m) \\
\xi & \text{otherwise}
\end{cases}
\]

\[
\langle p, q \rangle f = \begin{cases} \\
(p d_1, q d_2) & \text{if } f = (d_1, d_2), \; p d_1 \neq \xi \text{ and } q d_2 \neq \xi \\
\xi & \text{otherwise}
\end{cases}
\]

Employing the definition of (\( \langle \cdot, \cdot \rangle \)), it is straightforward to show that (\( ID \rightarrow ID \)) \( d = S d \) if \( d = (d_1, d_2) \). Similarly (\( ID \rightarrow ID \)) \( d = d \) for \( d \in D_m \Rightarrow c D_m \).

\( S \) is the projection that models strict demand; we can ask \( f \) uses its argument strictly when called in a strict context, just be asking whether

\[(ID \rightarrow S)f = (S \rightarrow S)f\]

Applying both sides to \( \bot \) we see

\[
\begin{align*}
(S \circ f) \bot &= (S \circ f \circ S) \bot \\
\implies S(f \bot) &= S(f(S \bot)) \\
\implies S(f \bot) &= S(f \xi) \\
\implies S(f \bot) &= S \xi \\
\implies S(f \bot) &= \xi \\
\implies f \bot &= \bot
\end{align*}
\]

so this condition does indeed imply that \( f \) is strict.
6.2 Conjunction of demands

It is often a case that strictness projections should be combined in a safe way employing additional context information. The following example illustrates this necessity.

Example 6.1. Let us consider a function \( g = \lambda(x,y) \to \ldots \) and for a fixed projection \( p \) we know that \( ((ID, ID) \ MEMORY \ p) g = ((q_x, q_y) \ MEMORY \ p) g \) for some \( q_x \) and \( q_y \). Assume now that for a function \( f = \lambda z \to g(z,z) \) we are interested in finding a non-trivial projection \( q_z \), such that

\[
(ID \ MEMORY \ p) f = (q_z \ MEMORY \ p) f
\]

by combining information from \( q_x \) and \( q_y \).

The example above leads to the following definition.

Definition 6.1 (Projection conjunction). Let \( p \) be a fixed projection. Then the operator \( \& \) is conjunction operator with respect to \( p \) iff for all \( g \in D_\infty \) and projections \( q_1, q_2 \)

\[
((ID, ID) \ MEMORY \ p) g = ((q_1, q_2) \ MEMORY \ p) g
\]

where \( \text{mult}_2 = \lambda d \to (d, d) \).

In the presence of an isomorphism between curried and uncurried functions, Definition 6.1 could be formulated in terms of functions of the form \( D_\infty \Rightarrow_c D_\infty \Rightarrow_c D_\infty \) rather than \( D_\infty \times D_\infty \Rightarrow_c D_\infty \). However, we preferred to do it for uncurried counterparts for the sake of clarity.

The \( \& \) operation is commutative, associative and idempotent. It distributes over \( \sqcup \) and has \( ID \) as unit.

It is obvious that the least upper bound operator \( \sqcup \) is a good (i.e., safe) candidate for the role of \& and, as \( ((q_1, q_2) \ MEMORY \ p) \sqsubseteq ((q_1 \sqcup q_2, q_1 \sqcup q_2) \ MEMORY \ p) \). However, in the case of strictness projections we can do slightly better if \( p = S \):

\[
(p_1 \& p_2) x = \begin{cases} 
\hat{x} & \text{if } p_1 x = \hat{x} \text{ or } p_2 x = \hat{x} \\
(p_1 x \sqcup (p_2 x) & \text{otherwise}
\end{cases}
\]

The correctness of the construction above with respect to Definition 6.1 is easy to show basing on the definition of the combinator \( \langle, \rangle \). The following lemma allows one to reuse the same construction for \& with respect to a family of projections, smaller than \( S \):

Lemma 6.1. If \( p \sqsubseteq S \) and \& is a conjunction operator with respect to \( S \) then \& is conjunction with respect to \( p \).

Proof. Straightforward by Lemma 4.1 and expanding \( p \) from Definition 6.1 to \( (p \circ S) \).
\[ \theta_1 \& \theta_2 = \{ [x \mapsto \theta_1(x) \& \theta_2(x) \mid x \in \text{dom}(\theta_1) \cup \text{dom}(\theta_2)], r_1 \& r_2 \} \]
where \( \theta_1 = (\phi_1, r_1) \) and \( \theta_2 = (\phi_2, r_1) \)

Fig. 4. \& operation for projection environments

Definition 6.1 of the conjunction of projections is crucial to construct compositional analyses that combine results from sub-cases. It is less obvious that the default projection environment \( \theta \) should be chosen in a way that its default projection \( r \) is a unit of the \& operation. *Ilya: Explain it better!*

6.3 A hyperstrict demand

We consider a distinguished projection \( H \), such that \( H \otimes d \equiv \otimes \) for any \( d \). It is convenient to model a hyperstrict demand. The demand \( H \) can be placed on the value \( d \) before \( S \) if evaluation of \( d \) is guaranteed to diverge. Therefore, the following equivalences are straightforward to demonstrate:

\[ \llbracket H, p \rrbracket = \llbracket p, H \rrbracket = \llbracket H, H \rrbracket = H \]
\[ H \Rightarrow p = p \Rightarrow H = H \]

6.4 Projection-based strictness analysis

In the remainder of this section we will refer to the identity projections \( ID \) as \( L \) (for “lazy”). A projection-based strictness analysis is defined formally in Figure 5. For the topmost expression the strictness analysis is typically initialized with a strict demand

\[ \overset{\text{L}}{\ldots} \overset{\text{L}}{\ldots} \overset{\text{S}}{\ldots} \]
where \( n \) is the arity of the expression.

In order to show the correctness of the strictness analysis, namely, restrict the set of input projections, so the safety theorem for strictness is formulated slightly different than for absence.

**Theorem 6.1.** The projection-based strictness analysis \( \mathcal{S}[\cdot] \) is safe with respect to the denotational semantics \([\cdot]\) for any input projection \( p \) such that either \( p = L \) or \( p \subseteq S \).
In particular, this restriction allowed us to replace \( \sqcup \) by \& in several cases according to Lemma 6.1 in order to improve the analysis precision.

**Proof.** The proof of the safety of the strictness analysis is mostly similar to the one of Theorem 5.1 except a few notable differences.

- The first clause of the analysis handles the case when a put projection \( p = L \). In this case the result of the analysis is trivial and returns a projection type \( (\theta \top, L) \) assuming that only trivial projections can be put on free variables and arguments of \( e \) (if there are any). It is important to realize that the demand \( p \) put in all other cases is less or equal than a strict demand \( S \) (i.e., \( p \subseteq S \)).
- Every time when a result projection environment \( \theta \) for \( e \) is obtained from two other, say, \( \theta_1 \) and \( \theta_2 \), resulting from analysis sub-expressions, each of which are evaluated
\[ \mathcal{S}[e]_\theta p \text{ takes an expression } e, \text{ a transformer environment } \hat{\rho} \text{ and a demand } p \text{ and computes a} \\
\text{demand type } \langle \theta, q \rangle \text{ for } e. \]

\[
\mathcal{S}[e]_\theta L = \langle \theta, L \rangle
\]

\[
\mathcal{S}[\lambda x \to e]_\theta p = \begin{cases} 
\langle \theta, q \rangle = \begin{cases} 
\hat{\rho}(x)(p) & \text{if } x \in \text{dom}[\hat{\rho}] \\
\langle \theta, L \rangle & \text{otherwise}
\end{cases} \\
in \langle \theta \setminus \{x\}, q \rangle \to q
\end{cases}
\]

\[
\mathcal{S}[\text{case } e \text{ as } z \text{ of } (x, y) \to a]_\theta p = \begin{cases} 
\langle \theta, q_a \rangle = \mathcal{S}[a]_\theta p \\
\langle \theta, \hat{\rho} \rangle = \mathcal{S}[e]_\theta \langle \theta_0(x), \theta_0(y) \rangle
\end{cases}
\]

\[
\mathcal{S}[\text{letrec } x = e_1 \text{ in } e_2]_\theta p = \begin{cases} 
p' = \begin{cases} 
L & \text{if } n = m = \text{arity}(P) \\
\left(\hat{\rho} \cup [\{x\} \to \langle \theta, H \rangle'] \right) & \text{otherwise}
\end{cases} \\
\hat{\rho}^m = \text{lfp}(\hat{\Theta}(\hat{\rho}, e_1, p')) \\
\langle \theta, q \rangle = \mathcal{S}[e_2]_\theta p
\end{cases}
\]

\[\text{Fig. 5. Strictness analysis}\]
in the process of evaluating $e$, $\theta_1$ and $\theta_2$ are combined via $\&$ operator as a better alternative for $\sqcup$ (since $p \subseteq S$ for all analysis clauses except the first one). The safety result is obtained by combining the definition of the operator $\sqcap$ with Definition 6.1 of the operator $\&$.

- $\theta_{\top}$ is used instead of $\theta_{\bot}$, as it provides a “default demand” $L = ID$, which is a unit for $\&$.
- If a demand put on a lambda-expression is not of the shape $(L \to p)$ or smaller, the body is analysed with a strictness demand $S$ in order to get non-trivial strictness demands on parameters and free variables. This is justified by the fact that $S$ is, indeed, a safe demand in the context of $S$.
- When analyzing an expression of the form $(\text{case } e \text{ as } x \text{ of } P_i y_i \to e_i)$, the scrutinized expression $e$ is analyzed with a strict demand $S$, which is justified by observation that it is safe to replace $[e]_p$ with $S([e]_p)$ if the demand $S$ or smaller is put on the whole case-expression.

7 Making Projection-Based Analysis Practical

Ilya: Describe representation of demands in GHC and intuition behind them

Ilya: Write down a section about result types that help to track bottoming functions

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.

7.1 Sums and products

In the presentation so far we have focused exclusively on product types, such as tuples and other single-constructor algebraic data types. But what about sum types, that is, algebraic data types with more than one constructor, such as Boolean or lists?

SLPJ: more to come here...

7.2 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.

7.3 Dealing with seq

TODO: Describe the head-strict and head-used demands.
7.4 Finding fixpoints

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

\[ f \mathord{\text{xs}} = [y + 1 | x \leftarrow \mathord{\text{xs}}, y \leftarrow h x] \]

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

\[
\begin{align*}
  f \mathord{[\ ]} &= \mathord{[\ ]} \\
  f \mathord{(x : \mathord{\text{xs}})} &= \text{let } g \mathord{[\ ]} = f \mathord{\text{xs}} \\
  & \text{let } g \mathord{(y : \mathord{\text{ys}})} = y + 1 : g \mathord{\text{ys}} \\
  & \text{in } g \mathord{(h x)}
\end{align*}
\]

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 (1994). (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 numbers to back this up. John: I like it! Of course, numbers would be good. Analysis times on some benchmarks with and without the optimisation?

7.5 Splitting \( \theta \)

*Ilya:* Bring this section up to the chosen terminology
### Table 1. The effect of the worker/wrapper split

<table>
<thead>
<tr>
<th>Program</th>
<th>Absence only</th>
<th>Strictness only</th>
<th>Absence + Strictness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocations</td>
<td>Run time</td>
<td>Allocations</td>
</tr>
<tr>
<td>anna</td>
<td>-0.0%</td>
<td>0.1%</td>
<td>-1.8%</td>
</tr>
<tr>
<td>atom</td>
<td>0.0%</td>
<td>1.1%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>boyer2</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-31.8%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>-0.1%</td>
<td>1.2%</td>
<td>-2.3%</td>
</tr>
<tr>
<td>comp.lab_zift</td>
<td>5.0%</td>
<td>0.0%</td>
<td>-2.6%</td>
</tr>
<tr>
<td>compress2</td>
<td>0.0%</td>
<td>2.8%</td>
<td>-12.5%</td>
</tr>
<tr>
<td>expert</td>
<td>-0.0%</td>
<td>-0.0%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>fibheaps</td>
<td>-0.1%</td>
<td>-0.0%</td>
<td>-15.5%</td>
</tr>
<tr>
<td>fulsom</td>
<td>-0.3%</td>
<td>-4.5%</td>
<td>-5.2%</td>
</tr>
<tr>
<td>hpg</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>knights</td>
<td>-0.0%</td>
<td>0.0%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>multiplier</td>
<td>-3.0%</td>
<td>0.0%</td>
<td>-10.9%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-16.8%</td>
</tr>
<tr>
<td>parser</td>
<td>-0.1%</td>
<td>0.0%</td>
<td>-2.1%</td>
</tr>
<tr>
<td>pic</td>
<td>-3.0%</td>
<td>0.0%</td>
<td>-5.7%</td>
</tr>
<tr>
<td>pretty</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>puzzle</td>
<td>0.0%</td>
<td>0.1%</td>
<td>16.6%</td>
</tr>
<tr>
<td>reptile</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>sphere</td>
<td>0.0%</td>
<td>0.0%</td>
<td>1.8%</td>
</tr>
<tr>
<td>symalg</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-17.8%</td>
</tr>
<tr>
<td>treejoin</td>
<td>0.0%</td>
<td>-1.8%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>x2n1</td>
<td>0.0%</td>
<td>0.0%</td>
<td>-81.2%</td>
</tr>
</tbody>
</table>

... and 60 more programs

### Summary of results

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>-4.5%</th>
<th>-95.0%</th>
<th>-9.5%</th>
<th>-95.0%</th>
<th>-16.2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>5.0%</td>
<td>4.3%</td>
<td>16.6%</td>
<td>15.7%</td>
<td>16.5%</td>
<td>3.2%</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>0.0%</td>
<td>0.1%</td>
<td>-12.3%</td>
<td>0.4%</td>
<td>-16.9%</td>
<td>-3.3%</td>
</tr>
</tbody>
</table>

### 8 Implementation and Evaluation

The demand analyzer, as described in the current work, is implemented in the Glasgow Haskell Compiler, version 7.6.

**TODO:** What else?

#### 8.1 Experimental results

In this section, we report on the comparison of the performance results of programs, compiled by four different modifications of the compiler:

1. A compiler, which did not employ the worker-wrapper transformation;
2. A compiler, employing the worker-wrapeer transformation, basing only on “flat” absence information (i.e., optimizing away non-used function arguments);
S. Peyton Jones et al.

3. A compiler, employing the worker-wraper transformation, basing only on strictness information;
4. A compiler, employing a full-fledged worker-wrapper split, which takes both strictness and absence information into the account.

We take the performance results of the programs, compiled by the first compiler, as a baseline, so the numbers we report are relative to it. The results are represented on Figure 1. The numbers are differences in percent with respect to the results obtained by the baseline compiler, performing no optimizations.

9 Related Work

In the work Making “Strictness” More Relevant, the authors refer to the property, captured by our call demands as to applicativeness. It would be interesting to compare our analysis and the mentioned type systems in terms of efficiency and expressiveness.

References

Theory and Practice of Demand Analysis in Haskell


A Proofs of the Analysis Safety Results

Theorem 5.1. The projection-based absence analysis \( \mathcal{A}[\cdot] \) is safe with respect to the denotational semantics \( \llbracket \cdot \rrbracket \).

Proof. Taking the constant absence demand \( p = A \), the proof becomes trivial, as the analysis is represented then by the first rule only and taking \( q = A \) and \( \theta = \theta_\bot \) is safe. For \( p \neq A \) the proof is by induction on the size of an analysed expression, since the analysis is defined compositionally.

Case \( \mathcal{A}[x] p \). Assume \( x \notin \text{dom}(\hat{\rho}) \), then the resulting demand type is \( \langle \theta', q \rangle \), where \( \theta' = \theta_\bot \cup [x \mapsto p] \) and \( q = U = \text{ID} \). So, taking \( \langle \ast, f_x \rangle = \sigma(x) \) (\( f \equiv \text{const} \)), we have \( p [x] \sigma = p f_x \). From another side \( p (q [x] (\sigma)(f_x)) = p \) \( f_x \) = \( p f_x \), so the statement of the theorem is true.

Otherwise, assume \( x \in \text{dom}(\hat{\rho}) \), \( \langle \rho_x, f_x \rangle = \sigma(x) \), \( \langle \theta, q \rangle = \hat{\rho}(x)(p) \) and \( \theta' = \theta \cup [x \mapsto p] \). Having in mind that \( \hat{\rho}(x) \) is a two-point demand transformer with a threshold demand \( p' \), consider two possibilities:

\( (p \sqsubseteq p') \) By condition \( \sigma \bowtie \hat{\rho} \), we have \( p (q f_x(\theta \bowtie \rho_x)) = p f_x \rho_x \). What we need to show is, in fact that \( p (\sigma(x)) = p (q ((\theta' \bowtie \sigma)(x))) \). We do it by showing two inclusions. First,

\[
\begin{align*}
p (q ((\theta' \bowtie \sigma)(x))) \\
\overset{\text{by definition of } (\theta' \bowtie \sigma)}{=} \llbracket q \rrbracket \\
\overset{\text{by definition of } \sigma}{= \llbracket q \rrbracket (f_x(\theta \bowtie \rho))}
\end{align*}
\]

\( \llbracket q \rrbracket \) since \( \theta'(x) \) is a projection

\[
\begin{align*}
p (q (f_x(\theta \bowtie \rho))) \\
\overset{\text{by definition of } \sigma}{= \llbracket q \rrbracket (f_x(\theta \bowtie \rho))}
\end{align*}
\]

\( \llbracket q \rrbracket \) since \( \theta' \) is a projection and \( f_x \) is monotone

\[
\begin{align*}
p (f_x(\rho)) \\
\end{align*}
\]

Conversely,
p \langle q (\theta' \triangledown \sigma)(x) \rangle \\
= \chi \text{ by definition of } (\theta' \triangledown \sigma) \chi \\
p \langle q (\theta'(x)) (f_s(\theta' @ p)) \rangle \\
\catastrophes \since \theta'(x) = \theta \triangledown_0 [x \mapsto p] \chi \\
\langle p \circ q \circ p \rangle (f_s(\theta' @ p)) \\
\catastrophes \just \sinsertion \additional \projection q \chi \\
\langle p \circ q \circ p \rangle (f_s(\theta' @ p)) \\
= \chi \text{ (p \circ q) is a projection } \chi \\
\langle p \circ q \rangle (f_s(\theta' @ p)) \\
\catastrophes \theta \catastrophes \theta' \chi \\
p \langle q (f_s(\theta' @ p_s)) \rangle \\
\catastrophes \by \condition \that \sigma \catastrophes \hat{\rho} \chi \\
p \langle f_s(p_s) \rangle \\
\catastrophes \by \definition \of \sigma \chi \\
p \langle \sigma(x) \rangle .

(p' \subset p) \text{ By the definition of a two-point demand transformer, we have } \hat{\rho}(x)(p) = \langle \theta_\tau, U \rangle ^{\hat{\sigma}} \text{ so the statement of the theorem follows straightforwardly.}

**Case** \( \lambda \ex x = e_1 \in e_2 \parallel p \). By the induction hypothesis, we have \( p' ([e_1]_{\sigma}) = p' (q_s [e_1]_{\theta, \nu \sigma}) \).

Our goal is to construct \( \sigma' \), such that

1. \( \sigma' \equiv \sigma \triangledown_1 [x \mapsto [e_1]_{\sigma}] \), and
2. \( \sigma' \catastrophes \hat{\rho}' \),

so the induction hypothesis could be applied to \( e_2 \). We define the required \( \sigma' \) as follows:

\[
\sigma' = \sigma \triangledown_1 [x \mapsto (\sigma|_{FV(e_1)}) \lambda \rho. [e_1]_\rho], \tag{A 1}
\]

where \( FV(e_1) \) is a set of free variables of \( e_1 \). It is straightforward to show that \( \sigma' = \sigma \triangledown_1 [x \mapsto [e_1]_{\sigma}] \) (just unfolding the definition of \( \sigma(x) \)), so our goal is to show that \( \sigma' \catastrophes \hat{\rho}' \). Since \( \sigma \catastrophes \hat{\rho} \), and \( \sigma' \) and \( \hat{\rho}' \) are different only in binding corresponding to \( x \), we need to show that for any \( p_s \),

\[
p_s (f_s(p_s)) = p_s (q_s (f_s (\theta'_s @ p_s))), \tag{A 2}
\]

where \( \langle \rho_s, f_s \rangle = \sigma'(x) \) and \( \langle \theta'_s, q_s \rangle = \hat{\rho}'(x)(p_s) \). Again, since \( \hat{\rho}'(x) \) is a two-point demand transformer with a threshold demand \( p' \), we consider two cases.

(p' \subset p_s) This case is trivial, since then \( \hat{\rho}'(x)(p_s) = \langle \theta_\tau, U \rangle \). Therefore, \( p_s (q_s (f_s (\theta'_s @ p_s))) = p_s (U (f_s (\theta_\tau @ p_s))) = p_s (f_s (\theta_\tau @ p_s)) \), and the case is done.

---

\(^{4}\text{In practice, we do slightly better by taking } \theta = \langle [y \mapsto U \mid y \in dom(p_s)] \rangle \text{ instead of } \theta_\tau. \text{ i.e., variables not participating in } p_s \text{ are turned into bottoms, which does not affect the semantics, but is crucial for the analysis precision in other cases, when demand environments are combined.}\)

\(^{5}\text{In fact, a safe refinement } ([y \mapsto U \mid y \in FV(e_1)], A) \text{ is returned instead of mere } \theta_\tau.\)
(\( p_x \subseteq p' \)) By the induction hypothesis and Lemma 4.2, we have \( p_x (\|e_1\|_{\sigma}) = p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})) \).

One can notice that \( p_x (\|e_1\|_{\sigma'}, \sigma_1) = p_x (\|e_1\|_{\sigma}) = p_x (f_s, p_x). \) So, taking the definition (A 1), the required equality (A 2) can be reformulated as

\[
p_x (\|e_1\|_{\sigma}) = p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})), \tag{A 3}
\]

which, in its order, by the induction hypothesis, can be reformulated as

\[
p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})) = p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})), \tag{A 4}
\]

In order to prove the equality (A 4), we notice, that point-wise \( \theta'_x \ni \sigma = \lambda y. (\theta'_x(y))(f_x (\theta'_x \rho)) \) and \( \theta'_x \ni \sigma = \lambda y. f_y (\theta'_x \rho) \), where \( (f_x, f_y) = \sigma(y) \). Since \( \theta'_x \ni \sigma \) is a projection, we have \( (\theta'_x \ni \sigma) = (\theta'_x \ni \sigma) \). Therefore, since \( [\ni] \) is monotone in its arguments,

\[
p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})) \subseteq p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})).
\]

But by the induction hypothesis, \( p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})) = p_x (q_s (\|e_1\|_{\sigma})) \) and \( \theta'_x \) is a projection, so

\[
p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})) = p_x (q_s (\|e_1\|_{\sigma})) \supseteq p_x (q_s (\|e_1\|_{\sigma'} \hat{\theta})),
\]

which proves the equality (A 4), and, hence, (A 2).

Finally, removing \( \{x\} \) from the component \( \theta \) of the result of \( (\alpha \ni e_2)_{\rho} \) is safe with respect to the theorem statement, since \( x \) is not free in \( (\text{let } x = e_1 \text{ in } e_2). \)

Case \( (\alpha \ni e_1 \ni e_2)_{\rho} \). By the induction hypothesis about the analysis safety, we have

\[
(U \ni p) (\|e_1\|_{\sigma}) = (q_a (p \cdot q_1)) (\|e_1\|_{\theta \ni \sigma}) \tag{A 5}
\]

\[
q_a (\|e_2\|_{\sigma}) = q_a (q_2 \ni e_2 \ni e_2 \ni \theta \ni \sigma). \tag{A 6}
\]

The demand \( q_2 \) can be omitted and safely replaced by \( U \) on. One can notice that if \( \theta = \theta_1 \ni \theta_2 \), then \( \theta_1 \ni \sigma \subseteq \theta \ni \sigma \) and \( \theta_2 \ni \sigma \subseteq \theta \ni \sigma \), so we can replace both \( (\theta \ni \sigma) \) and \( (\theta \ni \sigma) \) in equations (A 5) and (A 6) by \( (\theta \ni \sigma) \). By unfolding the operator \( \text{app} \) and applying equations above, we obtain \( p (\|e_1 \ni e_2\|_{\sigma}) = p (q_1 \ni e_1 \ni e_2 \ni \theta \ni \sigma) \), which is the desired result.

Case \( (\alpha \ni [P]_{\rho} \ni p) \). The proof for this case is based on the analysis of the auxiliary function \( PT \) and an observation that product constructors do not contain free variables, hence, the “empty” projection environment \( \theta_1 \) is safe. The non-conservative result of \( PT \) just puts the same projections on arguments as those put on components of the product.

Case \( (\alpha \ni [\lambda x \ni e]_{\rho} \ni p) \). If a put demand \( p \) is of the shape \( U \ni p' \), then the body is analysed with a demand \( p' \), otherwise it is analysed with the most conservative demand \( U \) in order to gather information about free variables. By the induction hypothesis, taking \( \sigma \), such that \( x \notin \sigma, \sigma' = \sigma \ni x \ni (\iota, d), d \equiv \text{const} \) (i.e., \( \sigma' \ni \sigma \)), one has \( p_r (\|e\|_{\sigma'}) = p_r (q (\|e\|_{\sigma} \ni \sigma')). \)
Therefore,

\[(U \rightarrow p_r) ([\lambda x \rightarrow e]_\sigma)\]

\[= \hat{\iota} \text{ by definition of } [\lambda x \rightarrow e]_\sigma \text{ and shape of } \sigma' \]

\[\text{Fun}(\lambda d. (p_r [e]_{\sigma \sqcup [x \mapsto \langle d \rangle]}))\]

\[= \hat{\iota} \text{ by the induction hypothesis} \]

\[\text{Fun}(\lambda d. (p_r (q \in \theta \cap \sigma \sqcup [x \mapsto \langle d \rangle]))))\]

\[= \hat{\iota} \text{ by taking } q_x = \theta(x) \text{ and definition of } \nabla \]

\[\text{Fun}(\lambda d. (p_r (q \in \theta \cap \sigma \sqcup [x \mapsto \langle d \rangle]))))\]

\[= \hat{\iota} \text{ by folding } \lambda d.(\ldots) \text{ and the definition of } (q \rightarrow p) \]

\[(U \rightarrow p_r)(q_x \rightarrow q) ([\lambda x \rightarrow e]_{\sigma \cap \theta \cap \sigma}) \]

---

**Case (\(\alpha' \text{[case e as z of } (x,y) \rightarrow a]_\beta \))**. Let us consider the case when

\[\llbracket \text{case e as z of } (x,y) \rightarrow a \rrbracket_\sigma \neq \bot,\]

otherwise the proof is trivial. So we have for any \(\sigma \triangleright \hat{\rho}\)

\[\llbracket e \rrbracket_\sigma \quad (d_x, d_y) \quad \llbracket \text{case e as z of } (x,y) \rightarrow a \rrbracket_\sigma \quad (d_x, d_y) \]

Let us take \(\sigma' = \sigma \sqcup \{x \mapsto \langle d_x, d_y \rangle, y \mapsto \langle d_x, d_y \rangle, z \mapsto \langle d_x, d_y \rangle\}\). It is straightforward to show that

- \(\overline{\sigma'} = \overline{\sigma} \sqcup \{x \mapsto d_x, y \mapsto d_y, z \mapsto (d_x, d_y)\}\)

- \(\sigma' \triangleright \hat{\rho}\)
Therefore, we have
\[ p \llbracket \text{case } e \text{ as of } (x, y) \to a \rrbracket_{σ} = \llbracket \text{by chosen } σ' \rrbracket \]
\[ p \llbracket a \rrbracket_{σ} = \llbracket \text{by the induction hypothesis } \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{by the definition of } V \text{ and chosen } σ' \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{by moving projections to the result of } \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{by the induction hypothesis for } e \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{taking } θ = θ_a \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{equivalent rewriting } \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{monotonicity } \rrbracket \]
\[ p \llbracket q_a ([a]_0 v σ) \rrbracket = \llbracket \text{by definition of the semantics } \rrbracket \]
\[ p \llbracket q_a \llbracket \text{case } e \text{ as of } (x, y) \to a \rrbracket_{σ} \rrbracket = \llbracket \text{over the results of } \rrbracket \]

**Case** \((σ \llbracket \text{letrec } x = e_1 \text{ in } e_2 \rrbracket_β p)\). The proof is similar to the previous case with just one product constructor, except that the safe approximation is taken as \(\sqcup\) over the results of analyses for all branches and the scrutinized expression \(e\) is analysed with a conservative demand \(U\), rather than a refined one.

**Case** \((σ \llbracket \text{letrec } x = e_1 \text{ in } e_2 \rrbracket_β p)\). The proof of this case is similar to the analysis of a non-recursive binding \(σ \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_β p\). However, it is essential to show that \(\hat{ρ}'' \triangleright_σ σ'\), where \(\hat{σ} = \llbracket E(x, e_1, θ) \rrbracket_σ\), so the result could be obtained by induction hypothesis, proceeding to the result of \(σ \llbracket e_2 \rrbracket_β p\). The required relation \(\hat{ρ}'' \triangleright_σ σ'\) is proved by establishing three facts:

1. We show that the relation \(\triangleright_σ\) is inductive (or chain-closed), i.e., for every chain \(σ_0 \subseteq σ_1 \subseteq \ldots \) and every chain \(\hat{ρ}_0 \subseteq \hat{ρ} \subseteq \ldots \) if \(σ_i \triangleright_σ \hat{ρ}_i\) for all \(i\) then \((\sqcup σ_i) \triangleright_σ (\sqcup \hat{ρ}_i)\) (Abadi, 2000);
2. We take appropriate \(σ_0 = σ \sqcup [x \mapsto \{\}, λρ.⊥]\) and \(\hat{ρ}_0 = \hat{ρ} \sqcup [x \mapsto \langle p', θ_0, A \rangle]\) (σ, \(\hat{ρ}\) are from the formulation of the theorem) and show that \(σ_0 \triangleright_σ \hat{ρ}_0\);
3. Finally we show that for every \(σ'\) and \(\hat{ρ}'\), if \(σ' \triangleright_σ \hat{ρ}'\) then
   \[ E(x, e_1, σ_0)(σ') \triangleright \hat{E}(x, e_1, \hat{ρ}_0, p')(\hat{ρ}'). \]
By combining 1–3, we obtain
\[ \sigma' \rightarrow \lambda \sigma'. \sigma \sqcap \{ x \rightarrow (\sigma', \lambda \rho, [e_1]_p) \} \] and \( \tilde{E}(x, e_1, \tilde{\rho}_0, p') \) is defined in Figure 3.

1. Let us consider special chains \( \sigma \) and \( \hat{\rho} \), such that for any \( i \) and \( y \neq x \)

\begin{itemize}
  \item \( \sigma_i(y) = \sigma_0(y) \);
  \item \( \hat{\rho}_i(y) = \tilde{\rho}_0(y) \);
\end{itemize}

and \( \sigma_i(x) = (\rho_i, f) \) for any \( i \), where \( f \) is the same for all \( i > 0 \). One can notice that the described chains \( \sigma, \hat{\rho} \) are exactly those, generated by \( E(x, e_1, \sigma_0) \) and \( \tilde{E}(x, e_1, \tilde{\rho}_0, p') \).

So, in fact we need to show only that for any \( p \)
\[
\left( (\sigma' \sqcup \tilde{\rho}) \right) = p \left( q^* f (\lambda \rho \sqcup (\sqcap \rho_i)) \right),
\]

where \( \langle \theta^*, q^* \rangle = (\sqcup \tilde{\rho}_i)(x) = \sqcup (\tilde{\rho}_i(x)) \) (which exists and is a projection thanks to Proposition 4.2) and \( (\rho_i, f) = \sigma_i(x) \) (since \( (\sqcup \sigma_i)(x) = \sqcup (\rho_i, f) \)).

What comes immediately is that \( \theta'^* \sqcup \tilde{\theta} \), for any \( i \) and therefore by \( \sigma_i \sqcup \tilde{\rho}_i \), for any \( i : p \left( f \rho_i \right) = p \left( q^* f (\lambda \rho \sqcup \rho_i) \right) \) (A 8)

Also, \( p, q^*, f \) and \( \theta^* \) are continuous, so is their composition. Therefore,
\[
\begin{align*}
P (q^* f (\lambda \rho \sqcup (\sqcap \rho_i))) &= \xi \text{ by continuity} \\
\sqcup (p (q^* f (\lambda \rho \sqcup \rho_i))) &= \xi \text{ by (A 8)} \\
\sqcup (p (f \rho_i)) &= \xi \text{ by (A 8)} \\
p (f (\sqcup \rho_i)) &= \xi \text{ by continuity}
\end{align*}
\]

2. Since we take
\[
\begin{align*}
\sigma_0 &= \sigma \sqcup \{ x \rightarrow \{ \}, \lambda \rho, \perp \} \\
\rho_0 &= \hat{\rho} \sqcup \{ x \rightarrow \{ p', \langle \theta_1, A \rangle \} \}.
\end{align*}
\]

by assumptions we already have \( \sigma \sqcup \tilde{\rho} \), and it is straightforward to show that \( p (\perp) = p (A (\perp)) = \perp \), for any \( p \sqsubseteq p' \), which delivers the result \( \sigma_0 = \tilde{\rho}_0 \).

3. We have \( \sigma_0 \sqcup \hat{\rho}_0, \rho' \sqcup \tilde{\rho}' \) and
\[
\begin{align*}
E(x, e_1, \sigma_0) &= \lambda \sigma', \sigma \sqcup \{ x \rightarrow (\sigma', \lambda \rho, [e_1]_p) \} \\
\tilde{E}(x, e_1, \tilde{\rho}_0, p') &= \lambda \tilde{\rho}', (\text{let } \theta_1, q_1) = (\lambda \rho, [e_1]_p, p') \\
&\quad \text{in } \tilde{\rho} \sqcup \tilde{\rho}' \{ x \rightarrow (p', \langle \theta_2, \{ x \}, q_2 \rangle) \} \}
\end{align*}
\]

It is straightforward to show that if \( \rho_0 = \sigma_0 \) and \( \rho' = \tilde{\sigma}' \) then \( E(x, e_1, \rho_0)(p') = \tilde{E}(x, e_1, \sigma_0)(\sigma') \).
By unfolding the definitions above, the proof of the fact
\[ E(x, e_1, \sigma_0)(\sigma') \Rightarrow E(x, e_1, \hat{\rho}_0, p')(\hat{\rho}') \]
is exactly the same as the proof of the relation \( \Rightarrow \) in the case \( (\mathcal{A} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\hat{\beta}} p) \) for a non-recursive \textbf{let}-binding.

\[ \square \]