Under consideration for publication in J. Functional Programming

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

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 analysi 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:

$lenFst :: ([a], b) \to Int$
$lenFst\ x = \textbf{case}\ x\ \textbf{of}\ \{\ (p, q) \to length\ p\ \}$

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

$\textbf{case}\ (g\ t)\ \textbf{of}\ \{\ (p, q) \to wlenFst\ p\ \}$

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 analyis 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:

$f :: (Int, Int) \to Int$
$f\ p = <rhs>$

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]

$f :: (Int, Int) \to Int$
$f\ p = \textbf{case}\ p\ \textbf{of}$
$\quad (a, b) \to wf\ a\ b$

[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!

$$wf :: Int \to Int \to Int$$
$$wf\ a\ b = \textbf{let}\ p = (a,b)\ \textbf{in} < rhs >$$

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:

$$f :: (Int, Int) \to Int$$
$$f\ p = \textbf{case}\ p\ \textbf{of}\ (a,b) \to wf\ a$$
$$wf :: Int \to Int$$
$$wf\ a = \textbf{let}\ p = (a, error\ \texttt{"abs"})\ \textbf{in} < rhs >$$

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 take it 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:

$$f :: (Int, Int) \to Int$$
$$f\ p = (\textbf{case}\ p\ \textbf{of}\ (a,b) \to a) + 1$$

Then the worker-wrapper transformation will produce:

$$f :: (Int, Int) \to Int$$
$$f\ p = \textbf{case}\ p\ \textbf{of}\ (a,b) \to wf\ a$$
$$wf :: Int \to Int$$
$$wf\ a = \textbf{let}\ p = (a, error\ \texttt{"Urk"})$$
$$\quad \textbf{in}\ (\textbf{case}\ p\ \textbf{of}\ (a,b) \to a) + 1$$

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:

$$wf :: Int \to Int$$
$$wf\ a = a + 1$$

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 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 $error :: String \to a$ 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:

$$f\ [\,]\ y = error\ \texttt{"urk"}$$
$$f\ (x:xs)\ y = y$$

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:

$$g_1\ x\ y = g_1\ y\ x$$
$$g_2\ x\ y = error\ x$$

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 this worker/wrapper split for g_2:

$$\begin{array}{l} g_2\ x\ y = wg_2 \\ wg_2 = \textbf{let}\ x = error\ \texttt{"abs"} \\ \qquad\qquad\ \ y = error\ \texttt{"abs"} \\ \qquad\ \textbf{in}\ \ error\ x \end{array}$$

This is obviously wrong, because the call *error x* will attempt to print the string x, but we have not passed x to 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 $seq :: a \to b \to b$, 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:

$$g3\ a\ b \not\equiv (\lambda x \to g3\ a\ b\ x)$$

The former is $\bot$, 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 (**?**).

$g4\ x = (\lambda y \to x)$ `` `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, *seq* can *evaluate* a function without *calling* it, and our analysis must recognise this distinction. (A sound alternative would be to treat *seq* as lazy in its first argument, but that would be foolish becaues programmers often use *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:

$f\ x\ y\ z = <rhs>$

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 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 analyis* accomodates 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 $(fst\ x + y)$ transforms the demand S on the result into a demand (A,S) on x, and S on y.

Similarly a function

$$f\ x\ y = <rhs>$$

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 contructors? 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.*

Types

$$\tau \quad ::= \quad P\,\overline{\tau} \mid \tau \to \tau \mid a \mid \forall a.\tau$$

Expressions

e	::=	x	Variables
	$\mid$	$e\ e$	Applications
	$\mid$	$\lambda x \to e$	Lambda-expressions
	$\mid$	P	Data constructors
	$\mid$	**let** $x = e$ **in** e	Let-bindings
	$\mid$	**letrec** $x = e$ **in** e	Letrec-bindings
	$\mid$	**case** e **as** x **of** $\overline{P\,\overline{y} \to e_P}$	Case-expressions

Constructor signatures

$$\Sigma \quad ::= \quad \emptyset \mid \Sigma, P^{\alpha} : \forall \overline{a}.\overline{\tau}^{\alpha} \to P\,\overline{a}$$

Fig. 1. Syntax of the Core language

$$\begin{aligned}
Env &= \mathsf{Var} \Rightarrow_c D_\infty \\
[\![e]\!] &: Env \Rightarrow_c D_\infty
\end{aligned}$$

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![e_1\ e_2]\!]_\rho &= \mathsf{app}([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![P]\!]_\rho &= \mathsf{Fun}(\lambda d_1 \ldots \lambda d_n.(d_1, \ldots, d_n)) \\
[\![\lambda x \to e]\!]_\rho &= \mathsf{Fun}(\lambda d.[\![e]\!]_{\rho'}) \ \text{ where } \rho' = \rho[x \mapsto d] \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_\rho &= [\![e_1]\!]_{\rho'} \text{ where } \rho' = \rho[x \mapsto [\![e_1]\!]_\rho] \\
[\![\textbf{letrec } x = e_1 \textbf{ in } e_2]\!]_\rho &= [\![e_2]\!]_{\mathsf{lfp}(\mathtt{E}(x,e,\rho))} \ \text{ where } \mathtt{E}(x,e,\rho) = \lambda\rho'.\rho \sqcup \rho'[x \mapsto [\![e]\!]_{\rho'}] \\
[\![\textbf{case } e \textbf{ as } x \textbf{ of } \overline{P\,\overline{y} \to e_P}]\!]_\rho &= \begin{cases} [\![e_P]\!]_{\rho[\overline{y \mapsto d}]} & \text{if } [\![e]\!]_\rho = P\,\overline{d} \text{ and } P \text{ is a } \textbf{case}\text{-branch} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

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 $\tau \to \tau$ or type variables a. Signatures Σ provide definitions of the datatype constructors P_α coupled with their arities α.

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 Env as an element of D_∞. Don't we rather define a two-argument function from expression and environment to D_∞?

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 ρ is an element of the domain D_∞.

Following Benton *et al.* (2009), D_∞ 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:

$$\begin{array}{rcll} F(D^-, D^+) & = & (& (D^- \Rightarrow_c D^+) \\ & + & & \prod_{\alpha_1} D^+ \qquad P_1^{\alpha_1} \in \Sigma \\ & + & & \ldots \qquad \ldots \\ & + & & \prod_{\alpha_k} D^+ \qquad P_k^{\alpha_k} \in \Sigma \\ & + & & 1_{bad} \quad)_\perp \end{array}$$

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 $D_\perp$ is *lifting* that instruments a domain with a new bottom element $\perp$.

Environments ρ are just continuous functions $\mathsf{Var} \Rightarrow_c D_\infty$ from syntactic variables to elements of the domain D_∞. *SLPJ: Really? I thought it was a partial function, and not in D_∞ 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_∞. Similarly, we construct elements of other summands using product constructors $P_i^{\alpha_i}$ as tags for embeddings $\underbrace{D_\infty \times \ldots \times D_\infty}_{\alpha_i} \Rightarrow_c D_\infty$. 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_Σ.*

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 $d \in D$,

$$\begin{array}{rcl} p\, d & \sqsubseteq & d \\ p\, (p\, d) & = & p\, d \end{array}$$

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):

$$\begin{array}{rcl} p & \sqsubseteq & ID = \lambda x.x \\ p \circ p & = & p \end{array}$$

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 $\perp$. 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 *fst*, by showing that

$$fst = fst \circ p$$

That is, you can apply p to *fst*'s argument to throw information away, and *fst* will not notice.

4.3 Properties of projections

We will refer to a set of all projections on a domain D as $Pr(D)$. Projections possess a number of useful properties. For instance, they form a complete lattice under the point-wise $\sqsubseteq$ ordering, with $\top = ID$ and $\bot = \lambda x \to \bot_D$. 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 $p_2 \sqsubseteq ID$, we have $p_1 \circ p_2 \sqsubseteq p_1 \circ ID = p_1$. □

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_∞

In our development we will be considering projections over the semantic domain D_∞. Four particular projections will be particularly useful:

- $ID = \lambda x.x$, 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_∞. Since q is a projection, we have $q \sqsubseteq ID$, therefore $f \circ q \sqsubseteq f$ (by monotonicity of f). Similarly, $p \sqsubseteq ID$ implies $\forall g.p \circ g \sqsubseteq g$. Taking $g = f \circ q$, we obtain $p \circ f \circ q \sqsubseteq f \circ q \sqsubseteq f$, i.e., $(p \to q) \sqsubseteq ID$. 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_∞ is established component-wise, so for any $g = (d_1,d_2)$, we have $(p,q)g = (p\ d_1, q\ d_2) \sqsubseteq (d_1,d_2) = g$. □

In the remainder of the paper we will consider projections on D_∞ 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 $BOT = \lambda x \to \bot$ and deriving higher-order projections using $(\cdot \to \cdot)$ and $(\cdot,\cdot)$ operators.

For instance, let us consider a function $f = \lambda x \to \lambda y \to x$. 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 $p = ID \to ID \to ID$. The answer for this particular example is easy to find, taking $q = ID \to BOT \to ID$.

4.5 Projection environments

Value environments as partial functions $\mathsf{Var} \rightharpoonup D_\infty$ 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 $\theta = \langle \phi, r \rangle$ where ϕ a *partial* function from variables to projections and a r is *default* projection. An application @ of a projection environment θ to a value environment ρ is defined point-wise as follows:

$$(\theta @ \rho)(x) = \begin{array}{ll} \phi(x)(\rho(x)) & \text{if } x \in dom(\phi) \\ r(\rho(x)) & \text{otherwise} \end{array}$$

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

$$\begin{array}{rcl} \theta_1 \sqcup \theta_2 & = & \langle [x \mapsto \theta_1(x) \sqcup \theta_2(x) \mid x \in dom(\theta_1)) \cup dom(\theta_2)], r_1 \sqcup r_2 \rangle \\ & & \text{where } \theta_1 = \langle \phi_1, r_1 \rangle \text{ and } \theta_2 = \langle \phi_2, r_1 \rangle \end{array}$$

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* $(Var \rightharpoonup D_\infty) \Rightarrow_c (Var \rightharpoonup D_\infty)$*).*

We use the notation $\theta \sqcup [x \mapsto p]$ for least upper bound of θ's ϕ 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 θ's ϕ.

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 $\widehat{\tau}$ is a pair $\langle \theta, q \rangle$ for some projection q and a projection environment θ.

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

Definition 4.4 (Projection transformers)**.** Projection transformer $\mathscr{T}$ 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

$$\begin{array}{l} g :: (Int, Int, Int) \to [a] \to (Int, Bool) \\ g\ (a, b, c) = \textbf{case}\ a\ \textbf{of} \\ \qquad\qquad 0 \to error\ \texttt{"urk"} \\ \qquad\qquad \_ \to \lambda y \to \textbf{case}\ b\ \textbf{of}\ 0 \to (c, null\ y) \\ \qquad\qquad\qquad\qquad\qquad\qquad \_ \to (c, False) \end{array}$$

a part of the projection transformer table will look as follows (projection environment components are omitted, as g does not contain free variables):

Put projection on g	Safe projection
ID	ID
$ID \to ID$	$ID \to ID$
$ID \to ID \to (BOT, ID)$	$(ID, ID, BOT) \to ID \to ID$
$ID \to ID \to (ID, BOT)$	$ID \to BOT \to ID$

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 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 $\widehat{\mathscr{T}}$ by taking a finite sequence of input projections $p_1 \sqsubset \ldots \sqsubset p_n$, such that the value $\mathscr{T}(p_i) = \widehat{\tau}_i = \widehat{\mathscr{T}}(p_i)$ is computed precisely, and for any p' we have

$$\begin{array}{lcll} \widehat{\mathscr{T}}(p') & = & \widehat{\tau}_i & \text{if } p_{i-1} \sqsubset p' \sqsubseteq p_i \\ \widehat{\mathscr{T}}(p') & = & \widehat{\tau}_0 & \text{if } p' \sqsubseteq p_0 \\ \widehat{\mathscr{T}}(p') & = & \top & \text{otherwise.} \end{array}$$

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

$$\begin{array}{lcll} \widehat{\mathscr{T}}(p') & = & \widehat{\tau} & \text{if } p' \sqsubseteq p \\ \widehat{\mathscr{T}}(p') & = & \top & \text{otherwise.} \end{array}$$

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 $(p, \widehat{\tau})$, where p is a "threshold" projection and $\widehat{\tau}$ is a projection type returned by the transformers for all $p' \sqsubseteq p$. The "update" operation for a transformer environment $\widehat{\rho}$, a variable x, a separator projection p and a projection type $\widehat{\tau}$ is denoted as $\widehat{\rho}\,[x \mapsto (p, \widehat{\tau})]$.

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 $\mathscr{P}[\![\cdot]\!]$ from expressions to projection transformers, such that if

$$\mathscr{P}[\![e]\!]\ p = \langle \theta, q \rangle,$$

then

$$p\ [\![e]\!]_\rho = p(q\ [\![e]\!]_{\theta @ \rho}).$$

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 ρ, which we want our analysis to take into account. Therefore, we need to pass this information to the analysis $\mathscr{P}[\![\cdot]\!]$ as an extra component, hence the following definition.

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

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 $\widehat{\rho} = \{y \mapsto (ID, \langle \theta_\perp, ID \rangle)\}$, as the projection environment $\theta_\perp$ does not indicate that x *should not be* "bottomed". Therefore, a correct environment would be $\widehat{\rho}' = \{y \mapsto (ID, \langle \{x \mapsto ID\}, ID \rangle)\}$. I.e., if a projection ID is put on y, it *unleashes* the projection at least as big as 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 ρ with a transformer environment $\widehat{\rho}$. However, it is problematic because of the θ-components in the results of $\widehat{\rho}$, which provide information about variables, whereas ρ 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 σ is an element of $\Sigma = \mathsf{Var} \rightharpoonup (Env \times (Env \Rightarrow_c D_\infty))$. The flattening operation $\overline{\sigma} : \Sigma \to Env$ on expanded environments is defined point-wise as follows:

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

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

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

$$\begin{array}{ll} p\ (f\ \rho) = p\ (q\ (f\ (\theta @ \rho))) & \text{if } x \in dom(\widehat{\rho}) \\ f \equiv const & \text{otherwise} \end{array}$$

where $\langle \rho, f \rangle = \sigma(x)$ and $\langle \theta, q \rangle = \widehat{\rho}(x)(p)$.

In words, Definition 4.7 means that the transformer $\widehat{\rho}(x)$ 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 $x \in dom(\sigma) \setminus dom(\widehat{\rho})$, f is a constant function. The later condition means that if no information about x is provided by $\widehat{\rho}$, its value in σ 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 $\mathscr{P}[\![\cdot]\!]_{\cdot}$ from expressions and transformer environments to projection transformers.

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

$$\mathscr{P}[\![e]\!]_{\widehat{\rho}}\ p = \langle \theta, q \rangle \implies p\ [\![e]\!]_{\overline{\sigma}} = p\ (q\ [\![e]\!]_{\theta \nabla \sigma}),$$

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

It is noteworthy that the *deep* application $(\theta \nabla \sigma)$ for any x restricts not only arguments ρ_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 $x \notin dom(\widehat{\rho})$ then f_x is a constant function, and by taking $(\theta \nabla \sigma)(x) = \theta(x)(f_{x\_})$, 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 ρ, which components of e and ρ are *unused* when computing a value $[\![e]\!]_\rho$?". For example, in the definition:

$$f\ x\ y = \textbf{if}\ x \equiv 0\ \textbf{then}\ f\ (x-1)\ y\ \textbf{else}\ x$$

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:

$$st\ p = \textbf{case}\ p\ \textbf{of}\ \{\,(x,y) \to x\}$$

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 *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.[3] 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 $p = U \to U \to U$ for a function of type $\tau_1 \to \tau_2 \to \tau_3$ means that the function is going to be applied to both arguments and its result is going to be used. Having $q = U \to A \to U$ 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 $\bot$) θ and put the absent demand A on the expression itself.

It is notable, that in the case $(\mathscr{A}[\![x]\!]_{\widehat{\rho}}\ p)$ when analyzing a variable x, such that $x \in dom(\widehat{\rho})$, but $p \sqsupseteq p'$, where p' is a *threshold* demand for a two-point transformer $\widehat{\rho}(x)$, the transformer $\widehat{\rho}(x)$ 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 $\theta = \langle [y \mapsto U \mid y \in FV(e)], A \rangle$ 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 $\mathscr{A}[\![\cdot]\!]$ with respect to the denotational semantics.

Theorem 5.1. *The projection-based absence analysis* $\mathscr{A}[\![\cdot]\!]$ *is* safe *with respect to the denotational semantics* $[\![\cdot]\!]$.

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. □

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 *BOT* projection as A (for "absent") and to *ID* as U (for "used").

$\mathscr{A}[\![e]\!]_{\widehat{\rho}}\ p$ takes an expression e, a transformer environment $\widehat{\rho}$ and a demand p and computes a demand type $\langle \theta, q \rangle$ for e.

$$
\begin{array}{rcl}
\mathscr{A}[\![e]\!]_{\widehat{\rho}}\ A & = & \langle \theta_\perp, A \rangle \\
\\
\mathscr{A}[\![x]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta, q \rangle & = & \begin{cases} \widehat{\rho}(x)(p) & \text{if } x \in \mathit{dom}(\widehat{\rho}) \\ \langle \theta_\perp, U \rangle & \text{otherwise} \end{cases} \\ \mathit{in} & \langle \theta \sqcup [x \mapsto p], q \rangle \end{array} \\
\\
\mathscr{A}[\![e_1\ e_2]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta_1, q_a \to q_1 \rangle & = & \mathscr{A}[\![e_1]\!]_{\widehat{\rho}}\ (U \to p) \\ & \langle \theta_2, q_2 \rangle & = & \mathscr{A}[\![e_2]\!]_{\widehat{\rho}}\ q_a \\ \mathit{in} & \langle \theta_1 \sqcup \theta_2, q_1 \rangle \end{array} \\
\\
\mathscr{A}[\![P]\!]_{\widehat{\rho}}\ p & = & PT(P, p) \\
\\
\mathscr{A}[\![\lambda x \to e]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & p_r & = & \mathit{if}\ p = (U \to p')\ \mathit{then}\ p'\ \mathit{else}\ U \\ & \langle \theta, q \rangle & = & \mathscr{A}[\![e]\!]_{\widehat{\rho}}\ p_r \\ & q_x & = & \theta(x) \\ \mathit{in} & \langle \theta \setminus \{x\}, q_x \to q \rangle \end{array} \\
\\
\mathscr{A}[\![\textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta_a, q_a \rangle & = & \mathscr{A}[\![a]\!]_{\widehat{\rho}}\ p \\ & \langle \theta_e, \_ \rangle & = & \mathscr{A}[\![e]\!]_{\widehat{\rho}}\ (\theta_a(x), \theta_a(y)) \\ \mathit{in} & \langle (\theta_a \setminus \{x,y,z\} \sqcup \theta_e), q_a \rangle \end{array} \\
\\
\mathscr{A}[\![\textbf{case}\ e\ \textbf{as}\ x\ \textbf{of}\ \overline{P_i\ \overline{y_i} \to e_i}]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \overline{\langle \theta_i, q_i \rangle} & = & \overline{\mathscr{A}[\![e_i]\!]_{\widehat{\rho}}\ p} \\ & \langle \theta_a, q_a \rangle & = & \langle \sqcup_i \overline{\theta_i \setminus \overline{y_i}}, \sqcup_i \overline{q_i} \rangle \\ & \langle \theta_e, \_ \rangle & = & \mathscr{A}[\![e]\!]_{\widehat{\rho}}\ U \\ \mathit{in} & \langle ((\theta_a \setminus \{x\}) \sqcup \theta_e), q_a \rangle \end{array} \\
\\
\mathscr{A}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & p' & = & \overbrace{U \to \ldots \to U}^{\mathit{arity}(e_1)} \to U \\ & \langle \theta_x, q_x \rangle & = & \mathscr{A}[\![e_1]\!]_{\widehat{\rho}}\ p' \\ & \widehat{\rho}' & = & \widehat{\rho} \sqcup [x \mapsto \langle p', \langle \theta_x \setminus \{x\}, q_x \rangle \rangle] \\ & \langle \theta, q \rangle & = & \mathscr{A}[\![e_2]\!]_{\widehat{\rho}'}\ p \\ \mathit{in} & \langle \theta \setminus \{x\}, q \rangle \end{array} \\
\\
\mathscr{A}[\![\textbf{letrec}\ x = e_1\ \textbf{in}\ e_2]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & p' & = & \overbrace{U \to \ldots \to U}^{\mathit{arity}(e_1)} \to U \\ & \widehat{\rho}' & = & \widehat{\rho} \sqcup [x \mapsto \langle p', \langle \theta_\perp, A \rangle \rangle] \\ & \widehat{\rho}'' & = & \mathsf{lfp}\left(\widehat{\mathrm{E}}(x, e_1, \widehat{\rho}', p')\right) \\ & \langle \theta, q \rangle & = & \mathscr{A}[\![e_2]\!]_{\widehat{\rho}''}\ p \\ \mathit{in} & \langle \theta \setminus \{x\}, q \rangle \end{array}
\end{array}
$$

$$\widehat{\mathrm{E}}(x, e, \widehat{\rho}, p) = \lambda \widehat{\rho}'.\left(\mathit{let}\ \langle \theta_x, q_x \rangle = (\mathscr{A}[\![e]\!]_{\widehat{\rho} \sqcup \widehat{\rho}'}\ p)\ \mathit{in}\ \widehat{\rho} \sqcup \widehat{\rho}'\,[x \mapsto (p, \langle \theta_x \setminus \{x\}, q_x \rangle)]\right)$$

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

$$
\begin{array}{rcll}
PT(P, \underbrace{U \to \ldots \to U}_{m} \to (p_1, \ldots, p_n)) & = & \langle \theta_\perp, p_1 \to \ldots \to p_n \to U \rangle & \text{if } n = m = \mathit{arity}(P) \\
& = & \langle \theta_\perp, U \rangle & \text{otherwise}
\end{array}
$$

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 $\lightning$ ("lightning bolt"), such that $\lightning \sqsubset \bot$ (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 $f\ \lightning = \lightning$. 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 $\lightning$. We can therefore model evaluating a value as a projection S, such that

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

I.e., evaluating $\bot$ leads to divergence.

In order to deal with a new bottom $\lightning$, we replace the projection combinators $(\cdot \to \cdot)$, $(\cdot,\cdot)$ with new ones, $(\cdot \twoheadrightarrow \cdot)$, $(\!(\cdot,\cdot)\!)$, *evaluating* their arguments, defined correspondingly:

$$(p \twoheadrightarrow q)f = \begin{cases} \lambda d.p(f\ (q\ d)) & \text{if } f \in (D_\infty \Rightarrow_c D_\infty) \\ \lightning & \text{otherwise} \end{cases}$$

$$(\!(p,q)\!)f = \begin{cases} (p\ d_1, q\ d_2) & \text{if } f = (d_1,d_2),\ p\ d_1 \neq \lightning \text{ and } q\ d_2 \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$$

Employing the definition of $(\!(\cdot,\cdot)\!)$, it is straightforward to show that $(\!(ID,ID)\!)\ d = S\ d$ if $d = (d_1,d_2)$. Similarly $(ID \twoheadrightarrow ID)d = d$ for $d \in D_\infty \Rightarrow_c D_\infty$.

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

$$(ID \twoheadrightarrow S)f = (S \twoheadrightarrow S)f$$

Applying both sides to $\bot$ we see

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

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 $g = \lambda(x,y) \to ...$, and for a fixed projection p we know that $((ID,ID) \twoheadrightarrow p)\ g = ((q_x,q_y) \twoheadrightarrow p)\ g$ 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

$$(ID \twoheadrightarrow p)f = (q_z \twoheadrightarrow p)f$$

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

$$\begin{array}{c}(\!(ID,ID)\!) \twoheadrightarrow p)g = ((\!(q_1,q_2)\!) \twoheadrightarrow p)g \\ \Longrightarrow (ID \twoheadrightarrow p)(g \circ \mathtt{mult}_2) = ((q_1 \,\&\, q_2) \twoheadrightarrow p)(g \circ \mathtt{mult}_2)\end{array}$$

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

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 &, as $(\!(q_1,q_2)\!) \twoheadrightarrow p) \sqsubseteq (\!(q_1 \sqcup q_2, q_1 \sqcup q_2)\!) \twoheadrightarrow p)$. However, in the case of strictness projections we can do slightly better if $p = S$:

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

The correctness of the construction above with respect to Definition 6.1 is easy to show basing on the definition of the combinator $(\!(\cdot,\cdot)\!)$. 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 $(p \circ S)$. □

Considering projection environments (Section 4.5) as projections (together with application operation @), we extend the operation & to them in an obvious way, as shown on Figure 4. Correctness of the definition of conjunction for projection environments as presented in Figure 4 follows from Definition 6.1 applied point-wise (i.e, considering q_1 and q_2 for each variable involved) for some $g \in ((\mathsf{Var} \rightharpoonup D_\infty) \times (\mathsf{Var} \rightharpoonup D_\infty)) \Rightarrow_c D_\infty$.

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

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 θ should be chosen in a way that it's 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 $H\ d \equiv \lightning$ 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:

$$(\!(H,p)\!) = (\!(p,H)\!) = (\!(H,H)\!) = H$$
$$H \twoheadrightarrow p = p \twoheadrightarrow H = H$$

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 $\overbrace{L \twoheadrightarrow \ldots \twoheadrightarrow L}^{n} \twoheadrightarrow S$, 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 $\mathscr{S}[\![\cdot]\!]$ is* safe *with respect to the denotational semantics $[\![\cdot]\!]$ for any input projection p such that either $p = L$ or $p \sqsubseteq S$. 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 $\langle \theta_\top, L \rangle$ 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., $p \sqsubseteq S$).
- Every time when a result projection environment θ for e is obtained from two other, say, θ_1 and θ_2, resulting from analysis sub-expressions, *each* of which are evaluated

$\mathscr{S}[\![e]\!]_{\widehat{\rho}}\ p$ takes an expression e, a transformer environment $\widehat{\rho}$ and a demand p and computes a demand type $\langle \theta, q \rangle$ for e.

$$
\begin{array}{rcl}
\mathscr{S}[\![e]\!]_{\widehat{\rho}}\ L & = & \langle \theta_\top, L \rangle \\
\\
\mathscr{S}[\![x]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta, q \rangle & = & \begin{cases} \widehat{\rho}(x)(p) & \text{if } x \in \mathit{dom}(\widehat{\rho}) \\ \langle \theta_\top, L \rangle & \text{otherwise} \end{cases} \\ \mathit{in} & \langle \theta \,\&\, [x \mapsto p], q \rangle & & \end{array} \\
\\
\mathscr{S}[\![e_1\ e_2]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta_1, q_a \to q_1 \rangle & = & \mathscr{S}[\![e_1]\!]_{\widehat{\rho}}\ (L \twoheadrightarrow p) \\ & \langle \theta_2, q_2 \rangle & = & \mathscr{S}[\![e_2]\!]_{\widehat{\rho}}\ q_a \\ \mathit{in} & \langle \theta_1 \,\&\, \theta_2, q_1 \rangle & & \end{array} \\
\\
\mathscr{S}[\![P]\!]_{\widehat{\rho}}\ p & = & PT(P, p) \\
\\
\mathscr{S}[\![\lambda x \to e]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & p_r & = & \mathit{if}\ p = (L \twoheadrightarrow p')\ \mathit{then}\ p'\ \mathit{else}\ S \\ & \langle \theta, q \rangle & = & \mathscr{S}[\![e]\!]_{\widehat{\rho}}\ p_r \\ & q_x & = & \theta(x) \\ \mathit{in} & \langle \theta \setminus \{x\}, q_x \to q \rangle & & \end{array} \\
\\
\mathscr{S}[\![\textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \langle \theta_a, q_a \rangle & = & \mathscr{S}[\![a]\!]_{\widehat{\rho}}\ p \\ & \langle \theta_e, \_ \rangle & = & \mathscr{S}[\![e]\!]_{\widehat{\rho}}\ (\!(\theta_a(x), \theta_a(y))\!) \\ \mathit{in} & \langle (\theta_a \setminus \{x,y,z\} \,\&\, \theta_e), q_a \rangle & & \end{array} \\
\\
\mathscr{S}[\![\textbf{case}\ e\ \textbf{as}\ x\ \textbf{of}\ \overline{P_i\ \overline{y_i} \to e_i}]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & \overline{\langle \theta_i, q_i \rangle} & = & \overline{\mathscr{S}[\![e_i]\!]_{\widehat{\rho}}\ p} \\ & \langle \theta_a, q_a \rangle & = & \langle \sqcup_i\, \overline{\theta_i \setminus \overline{y_i}}, \sqcup_i\, \overline{q_i} \rangle \\ & \langle \theta_e, \_ \rangle & = & \mathscr{S}[\![e]\!]_{\widehat{\rho}}\ S \\ \mathit{in} & \langle ((\theta_a \setminus \{x\}) \,\&\, \theta_e), q_a \rangle & & \end{array} \\
\\
\mathscr{S}[\![\textbf{letrec}\ x = e_1\ \textbf{in}\ e_2]\!]_{\widehat{\rho}}\ p & = & \begin{array}{lrcl} \mathit{let} & p' & = & \overbrace{L \twoheadrightarrow \ldots \twoheadrightarrow L}^{\mathit{arity}(e_1)} \twoheadrightarrow S \\ & \widehat{\rho}' & = & \widehat{\rho} \sqcup [x \mapsto \langle p', \langle \theta_\bot, H \rangle \rangle] \\ & \widehat{\rho}'' & = & \mathsf{lfp}\left(\widehat{\mathtt{E}}(x, e_1, \widehat{\rho}', p')\right) \\ & \langle \theta, q \rangle & = & \mathscr{S}[\![e_2]\!]_{\widehat{\rho}''}\ p \\ \mathit{in} & \langle \theta \setminus \{x\}, q \rangle & & \end{array}
\end{array}
$$

$\widehat{\mathtt{E}}(x, e, \widehat{\rho}, p)$ is a monotonic function defined in the same way as for absence (Figure 3).
$PT(P, p)$ takes a product constructor P and a demand p and returns an approximation of the constructor's demand type given that demand.

$$
\begin{array}{rcll}
PT(P, \underbrace{L \twoheadrightarrow \ldots \twoheadrightarrow L}_{m} \twoheadrightarrow (\!(p_1, \ldots, p_n)\!)) & = & \langle \theta_\top, p_1 \twoheadrightarrow \ldots \twoheadrightarrow p_n \to L \rangle & \text{if } n = m = \mathit{arity}(P) \\
& = & \langle \theta_\top, L \rangle & \text{otherwise}
\end{array}
$$

Fig. 5. Strictness analysis

in the process of evaluating e, θ_1 and θ_2 are combined via & operator as a better alternative for $\sqcup$ (since $p \sqsubseteq S$ for all analysis clauses except the first one). The safety result is obtained by combining the definition of the operator ∇ 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 $(L \twoheadrightarrow p)$ 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 $(\textbf{case}\ e\ \textbf{as}\ x\ \textbf{of}\ \overline{P_i\ \overline{y_i} \to e_i})$, the scrutinized expression e is analyzed with a strict demand S on, which is justified by observation that it is safe to replace $[\![e]\!]_\rho$ with $S([\![e]\!]_\rho)$ 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:

$$f\ xs = [y+1 \mid x \leftarrow xs, y \leftarrow h\ x]$$

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

$$\begin{array}{l} f\ [\,] = [\,] \\ f\ (x:xs) = \textbf{let}\ g\ [\,] = f\ xs \\ \qquad\qquad\qquad\quad g\ (y:ys) = y+1 : g\ ys \\ \qquad\qquad\quad \textbf{in}\ \ g\ (h\ x) \end{array}$$

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 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?

7.5 Splitting θ

Ilya: Bring this section up to the chosen terminology

Table 1. *The effect of the worker/wrapper split*

Program	Absence only		Strictness only		Absence + Strictness	
	Allocations	Run time	Allocations	Run time	Allocations	Run time
`anna`	-0.0%	0.1%	-1.8%	0.1%	-5.4%	0.1%
`atom`	0.0%	1.1%	-0.0%	0.0%	-0.1%	-0.8%
`boyer2`	0.0%	0.0%	-31.8%	0.0%	-31.8%	0.0%
`cacheprof`	-0.1%	1.2%	-2.3%	1.6%	-18.2%	1.2%
`comp_lab_zift`	5.0%	0.0%	-2.6%	0.0%	0.2%	0.0%
`compress2`	0.0%	2.8%	-12.5%	-0.7%	-32.7%	1.4%
`expert`	-0.0%	-0.0%	-0.3%	-0.0%	-11.3%	-0.0%
`fibheaps`	-0.1%	-0.0%	-15.5%	0.0%	-15.6%	0.0%
`fulsom`	-0.3%	-4.5%	-5.2%	-4.5%	-6.7%	-6.2%
`hpg`	0.0%	0.0%	-1.2%	0.1%	-11.4%	0.1%
`knights`	-0.0%	0.0%	-0.0%	1.4%	-2.8%	1.4%
`multiplier`	-3.0%	0.0%	-10.9%	0.0%	-16.2%	0.0%
`nucleic2`	0.0%	0.0%	-16.8%	0.1%	-20.0%	0.0%
`parser`	-0.1%	0.0%	-2.1%	0.0%	-19.5%	0.0%
`pic`	-3.0%	0.0%	-5.7%	0.0%	-7.4%	0.0%
`pretty`	0.0%	0.0%	-0.1%	-0.0%	-10.4%	-0.0%
`puzzle`	0.0%	0.1%	16.6%	0.1%	16.5%	0.1%
`reptile`	0.0%	0.0%	-0.2%	0.0%	-12.3%	0.0%
`sphere`	0.0%	0.0%	1.8%	0.0%	-8.4%	0.0%
`symalg`	0.0%	0.0%	-17.8%	0.0%	-45.5%	0.0%
`treejoin`	0.0%	-1.8%	-0.0%	9.5%	-24.7%	-7.7%
`x2n1`	0.0%	0.0%	-81.2%	0.0%	-81.2%	0.0%
... and 60 more programs						
Summary of results						
Min	-3.0%	-4.5%	-95.0%	-9.5%	-95.0%	-16.2%
Max	5.0%	4.3%	16.6%	15.7%	16.5%	3.2%
Geometric mean	0.0%	0.1%	-12.3%	0.4%	-16.9%	-3.3%

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);

3. A compiler, employing the worker-wrapeer 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

Abadi, Martín. (2000). $\top$-$\top$-closed relations and admissibility. *Mathematical structures in computer science*, **10**(3), 313–320.

Baker-Finch, Clement A., Glynn, Kevin, & Jones, Simon L. Peyton. (2004). Constructed product result analysis for Haskell. *Journal of functional programming*, **14**(2), 211–245.

Benton, Nick, Kennedy, Andrew, & Varming, Carsten. (2009). Some Domain Theory and Denotational Semantics in Coq. *Pages 115–130 of:* Berghofer, Stefan, Nipkow, Tobias, Urban, Christian, & Wenzel, Makarius (eds), *Proceedings of the 22nd International Conference Theorem Proving in Higher Order Logics (TPHOLs 2009)*. Lecture Notes in Computer Science, vol. 5674. Munich, Germany: Springer.

Cousot, Patrick, & Cousot, Radhia. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of:* Sethi, Ravi (ed), *of the fourth Annual ACM Symposium on Principles of programming languages*.

Henglein, Fritz. (1994). Iterative fixed point computation for type-based strictness analysis. *Pages 395–407 of:* Le Charlier, Baudouin (ed), *Static Analysis, First International Symposium, SAS'94*. Lecture Notes in Computer Science, vol. 864. Namur, Belgium: Springer-Verlag.

Henglein, Fritz, & Jørgensen, Jesper. 1994 (Jan.). Formally optimal boxing. *Pages 213–226 of:* Boehm, Hans-J. (ed), *of the 21st Annual ACM Symposium on Principles of programming languages*.

Kubiak, Ryszard, Hughes, John, & Launchbury, John. (1991). Implementing projection-based strictness analysis. *Pages 207–224 of:* Rogardt Heldal, Carsten Kehler Holst, Philip Wadler (ed), *Proceedings of the 1991 Glasgow Workshop on Functional Programming*.

Mycroft, Alan. (1980). The theory and practice of transforming call-by-need into call-by-value. *Pages 269–281 of:* Robinet, Bernard (ed), *Symposium on Programming*. Lecture Notes in Computer Science, vol. 83. Springer.

Peyton Jones, Simon L. (1996). Compiling Haskell by Program Transformation: A Report from the Trenches. *Pages 18–44 of:* Nielson, Hanne Riis (ed), *of the sixth European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1058. Linköping, Sweden: Springer-Verlag.

Scott, Dana S. (1981). *Lectures on a mathematical theory of computation*. Tech. rept. PRG19. Department of Computer Science, University of Oxford.

Wadler, Philip, & Hughes, R. John M. (1987). Projections for strictness analysis. *Pages 385–407 of:* Kahn, Gilles (ed), *Functional programming languages and computer architecture*. Lecture Notes in Computer Science, vol. 274. Portland, Oregon: Springer-Verlag.

A Proofs of the Analysis Safety Results

Theorem 5.1. The projection-based absence analysis $\mathscr{A}[\![\cdot]\!]$ is *safe* with respect to the denotational semantics $[\![\cdot]\!]$.

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 $\theta = \theta_\perp$ 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 ($\mathscr{A}[\![x]\!]_{\widehat{\rho}}\ p$)**.** Assume $x \notin dom(\widehat{\rho})$, then the resulting demand type is $\langle \theta', q \rangle$, where $\theta' = \theta_\perp \sqcup [x \mapsto p]$ and $q = U = ID$. So, taking $\langle \_, f_x \rangle = \sigma(x)$ ($f \equiv const$), we have $p\ [\![x]\!]_{\overline{\sigma}} = p\ f_x$. From another side $p\ (q\ [\![x]\!]_{\theta' \nabla \sigma}) = p\ (ID\ ((\theta'(x))(f_x))) = p\ (p\ f_x) = p\ f_x$, so the statement of the theorem is true.

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

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

$$
\begin{aligned}
& p\ (q\ ((\theta' \nabla \sigma)(x))) \\
= \ & \wr \text{ by defintion of } (\theta' \nabla \sigma) \int \\
& p\ (q\ ((\theta'(x))(f_x(\theta' @ \rho)))) \\
\sqsubseteq \ & \wr \text{ since } \theta'(x) \text{ is a projection} \int \\
& p\ (q\ (f_x(\theta' @ \rho))) \\
\sqsubseteq \ & \wr \text{ since } \theta' \text{ is a projection and } f_x \text{ is monotone} \int \\
& p\ (q\ (f_x\ \rho)) \\
\sqsubseteq \ & \wr q \text{ is a projection} \int \\
& p\ (f_x\ \rho) \\
= \ & \wr \text{ by definition of } \overline{\sigma} \int \\
& p\ (\overline{\sigma}(x))
\end{aligned}
$$

Conversely,

$$
\begin{array}{ll}
 & p\ (q\ ((\theta' \nabla \sigma)(x))) \\
= & \{\!|\ \text{by defintion of } (\theta' \nabla \sigma)\ |\!\} \\
 & p\ (q\ ((\theta'(x))(f_x(\theta' @ \rho)))) \\
\sqsupseteq & \{\!|\ \text{since } \theta'(x) = \theta \sqcup [x \mapsto p]\ |\!\} \\
 & (p \circ q \circ p)(f_x(\theta' @ \rho)) \\
\sqsupseteq & \{\!|\ \text{just inserting an additional projection } q\ |\!\} \\
 & (p \circ q \circ p \circ q)(f_x(\theta' @ \rho)) \\
= & \{\!|\ (p \circ q) \text{ is a projection}\ |\!\} \\
 & (p \circ q)(f_x(\theta' @ \rho)) \\
\sqsupseteq & \{\!|\ \theta \sqsubseteq \theta'\ |\!\} \\
 & p\ (q\ (f_x(\theta @ \rho_x))) \\
= & \{\!|\ \text{by condition that } \sigma \bowtie \widehat{\rho}\ |\!\} \\
 & p\ (f_x\ \rho_x) \\
= & \{\!|\ \text{by definition of } \overline{\sigma}\ |\!\} \\
 & p\ (\overline{\sigma}(x)).
\end{array}
$$

$(p' \sqsubset p)$ By the definition of a two-point demand transformer, we have $\widehat{\rho}(x)(p) = \langle \theta_\top, U \rangle$,[4] so the statement of the theorem follows straightforwardly.

Case $(\not\propto [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{\widehat{\rho}}\ p)$**.** By the induction hypothesis, we have $p'\ ([\![e_1]\!]_{\overline{\sigma}}) = p'\ (q_x\ [\![e_1]\!]_{\theta_x \nabla \sigma})$. Our goal is to construct σ', such that

1. $\overline{\sigma'} = \overline{\sigma}[x \mapsto [\![e_1]\!]_{\overline{\sigma}}]$, and
2. $\sigma' \bowtie \widehat{\rho}'$,

so the induction hypothesis could be applied to e_2. We define the required σ' as follows:

$$\sigma' = \sigma \sqcup [x \mapsto \langle \overline{\sigma}|_{FV(e_1)}, \lambda \rho . [\![e_1]\!]_\rho \rangle], \tag{A 1}$$

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

$$p_x\ (f_x\ \rho_x) = p_x\ (q_x\ (f_x\ (\theta'_x @ \rho_x))), \tag{A 2}$$

where $\langle \rho_x, f_x \rangle = \sigma'(x)$ and $\langle \theta'_x, q_x \rangle = \widehat{\rho}'(x)(p_x)$. Again, since $\widehat{\rho}'(x)$ is a two-point demand transformer with a threshold demand p', we consider two cases.

$(p' \sqsubseteq p_x)$ This case is trivial, since then $\widehat{\rho}'(x)(p_x) = \langle \theta_\top, U \rangle$.[5] Therefore, $p_x\ (q_x\ (f_x\ (\theta'_x @ \rho_x))) = p_x\ (U\ (f_x\ (\theta_\top @ \rho_x))) = p_x\ (f_x\ \rho_x)$, and the case is done.

[4] In practice, we do slightly better by taking $\theta = \langle [y \mapsto U \mid y \in dom(\rho_x)], A \rangle$ instead of $\theta_\top$. I.e., variables not participating in ρ_x 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] In fact, a safe refinement $\langle [y \mapsto U \mid y \in FV(e_1)], A \rangle$ is returned instead of mere $\theta_\top$.

($p_x \sqsubseteq p'$) By the induction hypothesis and Lemma 4.2, we have $p_x\ (\llbracket e_1 \rrbracket_{\overline{\sigma}}) = p_x\ (q_x\ \llbracket e_1 \rrbracket_{\theta'_x \nabla \sigma})$. One can notice that $p_x\ (\llbracket e_1 \rrbracket_{\overline{\sigma}|_{FV(e_1)}}) = p_x\ (\llbracket e_1 \rrbracket_{\overline{\sigma}}) = p_x\ (f_x\ \rho_x)$. So, taking the definition (A 1), the required equality (A 2) can be reformulated as

$$p_x\ (\llbracket e_1 \rrbracket_{\overline{\sigma}}) = p_x\ (q_x\ \llbracket e_1 \rrbracket_{\sigma'_x @ \overline{\sigma}}), \tag{A 3}$$

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

$$p_x\ (q_x\ \llbracket e_1 \rrbracket_{\theta'_x \nabla \sigma}) = p_x\ (q_x\ \llbracket e_1 \rrbracket_{\sigma'_x @ \overline{\sigma}}), \tag{A 4}$$

In order to prove the equality (A 4), we notice, that point-wise $\theta'_x \nabla \sigma = \lambda y.(\theta'_x(y))(f_y\ (\theta'_x @ \rho_y))$ and $\theta'_x @ \overline{\sigma} = \lambda y.f_y\ (\theta'_x @ \rho_y)$, where $\langle \rho_y, f_y \rangle = \sigma(y)$. Since $\theta'_x(y)$ is a projection, we have $(\theta'_x \nabla \sigma) \sqsubseteq \theta'_x @ \overline{\sigma}$ and therefore, since $\llbracket \cdot \rrbracket$ is monotone in its arguments,

$$p_x\ (q_x\ \llbracket e_1 \rrbracket_{\theta'_x \nabla \sigma}) \sqsubseteq p_x\ (q_x\ \llbracket e_1 \rrbracket_{\sigma'_x @ \overline{\sigma}}).$$

But by the induction hypothesis, $p_x\ (q_x\ \llbracket e_1 \rrbracket_{\theta'_x \nabla \sigma}) = p_x\ (q_x\ \llbracket e_1 \rrbracket_{\overline{\sigma}})$ and θ'_x is a projection, so

$$p_x\ (q_x\ \llbracket e_1 \rrbracket_{\theta'_x \nabla \sigma}) = p_x\ (q_x\ \llbracket e_1 \rrbracket_{\overline{\sigma}}) \sqsupseteq p_x\ (q_x\ \llbracket e_1 \rrbracket_{\sigma'_x @ \overline{\sigma}}),$$

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

Finally, removing $\{x\}$ from the component θ of the result of $(\mathscr{A} \llbracket e_2 \rrbracket_{\widehat{\rho}'}\ p)$ is safe with respect to the theorem statement, since x is not free in (**let** $x = e_1$ **in** e_2).

Case $(\mathscr{A} \llbracket e_1\ e_2 \rrbracket_{\widehat{\rho}}\ p)$**.** By the induction hypothesis about the analysis safety, we have

$$(U \to p)\ (\llbracket e_1 \rrbracket_{\overline{\sigma}}) = (q_a \to (p \circ q_1))(\llbracket e_1 \rrbracket_{\theta_1 \nabla \sigma}) \tag{A 5}$$

$$q_a\ (\llbracket e_2 \rrbracket_{\overline{\sigma}}) = q_a\ (q_2\ \llbracket e_2 \rrbracket_{\theta_2 \nabla \sigma}). \tag{A 6}$$

The demand q_2 can be omitted and safely replaced by U.On can notice that if $\theta = \theta_1 \sqcup \theta_2$, then $\theta_1 \nabla \sigma \sqsubseteq \theta \nabla \sigma$ and $\theta_2 \nabla \sigma \sqsubseteq \theta \nabla \sigma$, so we can replace both $(\theta_1 \nabla \sigma)$ and $(\theta_2 \nabla \sigma)$ in equations (A 5) and (A 6) by $(\theta \nabla \sigma)$. By unfolding the operator app and applying equations above, we obtain $p(\llbracket e_1\ e_2 \rrbracket_{\overline{\sigma}}) = p\ (q_1\ \llbracket e_1\ e_2 \rrbracket_{\theta \nabla \sigma})$, which is the desired result.

Case $(\mathscr{A} \llbracket P \rrbracket_{\widehat{\rho}}\ p)$**.** 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 $\theta_\perp$ is safe. The non-conservative result of *PT* just puts the same projections on arguments as those put on components of the product.

Case $(\mathscr{A} \llbracket \lambda x \to e \rrbracket_{\widehat{\rho}}\ p)$**.** If a put demand p is of the shape $U \to p'$, 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 σ, such that $x \notin \sigma$, $\sigma' = \sigma \sqcup [x \mapsto \langle \_, d \rangle]$, $d \equiv const$ (i.e., $\sigma' \bowtie \widehat{\rho}$), one has $p_r\ (\llbracket e \rrbracket_{\overline{\sigma'}}) = p_r\ (q\ \llbracket e \rrbracket_{\theta \nabla \sigma'})$.

Therefore,

$$\begin{aligned}
& (U \to p_r)\ (\llbracket \lambda x \to e \rrbracket_{\overline{\sigma}}) \\
= \ & \wr \text{ by definition of } \llbracket \lambda x \to e \rrbracket_{\overline{\sigma}} \text{ and shape of } \sigma' \int \\
& \mathsf{Fun}(\lambda d.(p_r\ \llbracket e \rrbracket_{\overline{\sigma \sqcup [x \mapsto \langle _, d \rangle]}})) \\
= \ & \wr \text{ by the induction hypothesis } \int \\
& \mathsf{Fun}\left(\lambda d.(p_r\ (q\ \llbracket e \rrbracket_{\theta \nabla (\sigma \sqcup [x \mapsto \langle _, d \rangle])}))\right) \\
= \ & \wr \text{ by taking } q_x = \theta(x) \text{ and definition of } \nabla \int \\
& \mathsf{Fun}\left(\lambda d.(p_r\ (q\ \llbracket e \rrbracket_{((\theta \setminus \{x\}) \nabla \sigma) \sqcup [x \mapsto q_x(d)]}))\right) \\
= \ & \wr \text{ by folding } \lambda d.(\ldots) \text{ and the defitinion of } (q \to p) \int \\
& (U \to p_r)(q_x \to q)\ \left(\llbracket \lambda x \to e \rrbracket_{(\theta \setminus \{x\}) \nabla \sigma}\right).
\end{aligned}$$

Case ($\mathscr{A}[\![$**case** e **as** z **of** $(x,y) \to a]\!]_{\widehat{\rho}}\ p$)**.** Let us consider the case when

$$\llbracket \textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a \rrbracket_{\overline{\sigma}} \neq \bot,$$

otherwise the proof is trivial. So we have for any $\sigma \bowtie \widehat{\rho}$

$$\begin{aligned}
\llbracket e \rrbracket_{\overline{\sigma}} &= (d_x, d_y) \\
\llbracket \textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a \rrbracket_{\overline{\sigma}} &= \llbracket a \rrbracket_{\overline{\sigma} \sqcup [x \mapsto d_x, y \mapsto d_y, z \mapsto (d_x, d_y)]}
\end{aligned}$$

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

- $\overline{\sigma'} = \overline{\sigma} \sqcup [x \mapsto d_x, y \mapsto d_y, z \mapsto (d_x, d_y)]$
- $\sigma' \bowtie \widehat{\rho}$

Therefore, we have

$$
\begin{aligned}
& p\ [\![\textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a]\!]_{\overline{\sigma}} \\
= \ & \wr \text{ by chosen } \sigma' \int \\
& p\ [\![a]\!]_{\overline{\sigma'}} \\
= \ & \wr \text{ by the induction hypothesis } \int \\
& p\ (q_a\ ([\![a]\!]_{\theta_a \nabla \sigma'})) \\
= \ & \wr \text{ by the defintion of } \nabla \text{ and chosen } \sigma' \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta_a \nabla \sigma) \sqcup [x \mapsto (\theta_a(x))(d_x), y \mapsto (\theta_a(y))(d_y), z \mapsto (\theta_a(x), \theta_a(y))(d_x, d_y)]})), \text{ where } (d_x, d_y) = [\![e]\!]_{\overline{\sigma}} \\
= \ & \wr \text{ by moving projections to the result of } [\![e]\!]_{\overline{\sigma}} \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta_a \nabla \sigma) \sqcup [x \mapsto d'_x, y \mapsto d'_y, z \mapsto (d'_x, d'_y)]})), \text{ where } (d'_x, d'_y) = (\theta_a(x), \theta_a(y))[\![e]\!]_{\overline{\sigma}} \\
= \ & \wr \text{ by the induction hypothesis for } e \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta_a \nabla \sigma) \sqcup [x \mapsto d'_x, y \mapsto d'_y, z \mapsto (d'_x, d'_y)]})), \text{ where } (d'_x, d'_y) = (\theta_a(x), \theta_a(y))[\![e]\!]_{\theta_e \nabla \sigma} \\
= \ & \wr \text{ taking } \theta = \theta_a \sqcup \theta_e \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta \nabla \sigma) \sqcup [x \mapsto d'_x, y \mapsto d'_y, z \mapsto (d'_x, d'_y)]})), \text{ where } (d'_x, d'_y) = (\theta_a(x), \theta_a(y))[\![e]\!]_{\theta \nabla \sigma} \\
= \ & \wr \text{ equivalent rewriting } \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta \nabla \sigma) \sqcup [x \mapsto (\theta_a(x))(d_x), y \mapsto (\theta_a(y))(d_y), z \mapsto (\theta_a(x), \theta_a(y))(d_x, d_y)]})), \\
& \text{where } (d_x, d_y) = [\![e]\!]_{\theta \nabla \sigma} \\
= \ & \wr \text{ monotonicity } \int \\
& p\ (q_a\ ([\![a]\!]_{(\theta \nabla \sigma) \sqcup [x \mapsto d_x, y \mapsto d_y, z \mapsto (d_x, d_y)]})), \text{ where } (d_x, d_y) = [\![e]\!]_{\theta \nabla \sigma} \\
= \ & \wr \text{ by definition of the semantics } \int \\
& p\ (q_a\ [\![\textbf{case}\ e\ \textbf{as}\ z\ \textbf{of}\ (x,y) \to a]\!]_{(\theta \setminus \{x,y,z\}) \nabla \sigma})
\end{aligned}
$$

Case ($\mathscr{A}[\![\textbf{case}\ e\ \textbf{as}\ x\ \textbf{of}\ \overline{P_i\ \overline{y_i} \to e_i}]\!]_{\widehat{\rho}}\ p$)**.** 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 ($\mathscr{A}[\![\textbf{letrec}\ x = e_1\ \textbf{in}\ e_2]\!]_{\widehat{\rho}}\ p$)**.** The proof of this case is similar to the analysis of a non-recursive binding $\mathscr{A}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]_{\widehat{\rho}}\ p$. However, it is essential to show that $\widehat{\rho}'' \bowtie \sigma'$, where $\overline{\sigma'} = \mathsf{lfp}(\mathtt{E}(x, e, \overline{(\sigma)}))$, so the result could be obtained by induction hypothesis, proceeding to the result of $\mathscr{A}[\![e_2]\!]_{\widehat{\rho}''}\ p$. The required relation $\widehat{\rho}'' \bowtie \sigma'$ is proved by establishing three facts:

1. We show that the relation $\bowtie$ is *inductive* (or *chain-closed*), i.e., for every chain $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ and every chain $\widehat{\rho}_0 \sqsubseteq \widehat{\rho} \sqsubseteq \ldots$ if $\sigma_i \bowtie \widehat{\rho}_i$ for all i then $(\sqcup \sigma_i) \bowtie (\sqcup \widehat{\rho}_i)$ (Abadi, 2000);
2. We take appropriate $\sigma_0 = \sigma \sqcup [x \mapsto \langle \{\}, \lambda \rho . \bot \rangle]$ and $\widehat{\rho}_0 = \widehat{\rho} \sqcup [x \mapsto \langle p', \langle \theta_\bot, A \rangle \rangle]$ (σ, $\widehat{\rho}$ are from the formulation of the theorem) and show that $\sigma_0 \bowtie \widehat{\rho}_0$;
3. Finally we show that for every σ' and $\widehat{\rho}'$, if $\sigma' \bowtie \widehat{\rho}'$ then

$$\mathtt{E}(x, e_1, \sigma_0)(\sigma') \bowtie \widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p')(\widehat{\rho}'),$$

where $\mathtt{E}(x, e_1, \sigma_0) = \lambda\sigma'.\sigma \sqcup [x \mapsto \langle \overline{\sigma'}, \lambda\rho.[\![e_1]\!]_\rho \rangle]$[6] and $\widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p')$ is defined in Figure 3.

By combining 1–3, we obtain

$$\begin{array}{rcccl} \sigma' & = & \mathsf{lfp}\ \mathtt{E}(x, e_1, \sigma_0) & = & \sqcup_i \mathtt{E}^i(x, e_1, \sigma_0) \\ \widehat{\rho}'' & = & \mathsf{lfp}\ \widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p') & = & \sqcup_i \widehat{\mathtt{E}}^i(x, e_1, \widehat{\rho}_0, p'), \end{array}$$

such that $\sigma' \bowtie \widehat{\rho}''$, which is the desired statement.

1. Let us consider special chains σ_i and $\widehat{\rho}_i$, such that for any i and $y \neq x$
   - $\sigma_i(y) = \sigma_0(y)$;
   - $\widehat{\rho}_i(y) = \widehat{\rho}_0(y)$;

   and $\sigma_i(x) = \langle \rho_i, f \rangle$ for any i, where f is the same for all $i > 0$. One can notice that the described chains σ_i, $\widehat{\rho}_i$ are exactly those, generated by $\mathtt{E}(x, e_1, \sigma_0)$ and $\widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p')$. So, in fact we need to show only that for any p

   $$p\ (f\ (\sqcup \rho_i)) = p\ (q^*\ (f\ (\theta^* @ (\sqcup \rho_i)))), \tag{A 7}$$

   where $\langle \theta^*, q^* \rangle = (\sqcup \widehat{\rho}_i)(x) = \sqcup(\widehat{\rho}_i(x))$ (which exists and is a projection thanks to Proposition 4.2) and $\langle \rho_i, f \rangle = \sigma_i(x)$ (since $(\sqcup \sigma_i)(x) = \sqcup(\sigma_i)(x) = \langle \sqcup \rho_i, f \rangle$).
   What comes immediately is that $\theta^* \sqsupseteq \theta_i$ for any i and therefore by $\sigma_i \bowtie \widehat{\rho}_i$, for any i:

   $$p\ (f\ \rho_i) = p\ (q^*\ (f\ (\theta^* @ \rho_i))) \tag{A 8}$$

   Also, p, q^*, f and θ^* are continuous, so is their composition. Therefore,

   $$\begin{array}{ll} & p\ (q^*\ (f\ (\theta^* @ (\sqcup \rho_i)))) \\ = & \wr \text{ by continuity } \int \\ & \sqcup (p\ (q^*\ (f\ (\theta^* @ \rho_i)))) \\ = & \wr \text{ by (A 8) } \int \\ & \sqcup (p\ (f\ \rho_i)) \\ = & \wr \text{ by continuity } \int \\ & p\ (f\ (\sqcup \rho_i)) \end{array}$$

2. Since we take

   $$\begin{array}{rcl} \sigma_0 & = & \sigma \sqcup [x \mapsto \langle \{\}, \lambda\rho.\bot \rangle] \\ \widehat{\rho}_0 & = & \widehat{\rho} \sqcup [x \mapsto \langle p', \langle \theta_\bot, A \rangle \rangle], \end{array}$$

   by assumptions we already have $\sigma \bowtie \widehat{\rho}$, and it is straightforward to show that $p\ (\bot) = p\ (A\ (A\ \bot)) = \bot$, for any $p \sqsubseteq p'$, which delivers the result $\sigma_0 = \widehat{\rho}_0$.
3. We have $\sigma_0 \bowtie \widehat{\rho}_0$, $\sigma' \bowtie \widehat{\rho}'$ and

   $$\begin{array}{rcl} \mathtt{E}(x, e_1, \sigma_0) & = & \lambda\sigma'.\sigma \sqcup [x \mapsto \langle \overline{\sigma'}, \lambda\rho.[\![e_1]\!]_\rho \rangle] \\ \widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p') & = & \lambda\widehat{\rho}'.(\mathit{let}\ \langle \theta_x, q_x \rangle = (\mathscr{A}[\![e_1]\!]_{\widehat{\rho}_0 \sqcup \widehat{\rho}'}\ p') \\ & & \quad \mathit{in}\ \ \widehat{\rho} \sqcup \widehat{\rho}'[x \mapsto (p', \langle \theta_x \setminus \{x\}, q_x \rangle)]) \end{array}$$

[6] It is straightforward to show that if $\rho_0 = \overline{\sigma_0}$ and $\rho' = \overline{\sigma'}$ then $\mathtt{E}(x, e_1, \rho_0)(\rho') = \overline{\mathtt{E}(x, e_1, \sigma_0)(\sigma')}$.

By unfolding the definitions above, the proof of the fact

$$\mathtt{E}(x, e_1, \sigma_0)(\sigma') \bowtie \widehat{\mathtt{E}}(x, e_1, \widehat{\rho}_0, p')(\widehat{\rho}')$$

is exactly the same as the proof of the relation $\bowtie$ in the case $(\mathscr{A}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{\widehat{\rho}}\ p)$ for a non-recursive **let**-binding.

□