PERs from Partial Projections for Higher-order Polymorphic Demand Analysis

Kei Davis
Los Alamos National Laboratory
kei at lanl.gov

February 21, 2017

Abstract

First-order projection-based strictness analysis (properly, *demand analysis*) was first introduced by Wadler and Hughes in 1987 [?]. This development was restricted to first-order, monomorphic languages and provided no proof of soundness. Subsequent developments extended the theory to other analyses, approaches to higher order, and polymorphism. Yet other developments were based on generalization of projections, specifically partial equivalence relations and partial projections, the latter notably simplifying a difficulty with projection-based strictness dating back to Wadler and Hughes. However, to date no development has combined the desirable contributions of all of these approaches. We present the first practical higher-order, polymorphic, partial-projection/partial-equivalence-relation based demand analysis, together with a proof of satisfaction of the fundamental safety condition for such analyses. Such an analysis can be used both to improve serial code generation, as well as enable safe automatic parallelization, for lazy or non-strict functional languages.

1 Introduction and Background

We take as given that demand analysis is essential for good serial code generation, and for identifying where parallelism may be safely introduced. Numerous such analysis techniques have been proposed and implemented; we greatly narrow the field of discussion by restricting attention to those for which there is a formally stated notion of correctness that the technique has been proven to satisfy.

Analysis techniques can usually be identified as being based on either a non-standard denotational semantics or a non-standard typing. Examples in the latter category include those of Gomard [Go92], Jensen [Jen92], the Nielsons [NN88], Schmidt [Sch88], and Henglein and Mossin [HM94]. Our focus is on those techniques based on non-standard interpretation, in particular, those using projections or partial equivalence relations (PERs) as the basic abstract values.

A domain *projection* is a continuous idempotent function that approximates the identity. Launchbury [Lau88] hit upon the idea of using projections to encode degrees of staticness of data. The basic idea is that a projection maps to $\perp$ that part of a data structure that is dynamic (possibly not determined), and acts as the identity on that part which is static (definitely determined). Examples are the identity ID, the greatest projection, which specifies that values are entirely static; the constant $\perp$ function BOT, the least projection, which specifies that values are entirely dynamic; and projections FST and SND on product domains, defined by

$$FST\ (x, y) = (x, \perp)\ , \qquad\qquad SND\ (x, y) = (\perp, y)\ ,$$

specifying staticness in the first and second components of pairs, respectively. The nominal goal of analysis is, given function f denoted by some programming-language expression, and projection δ encoding the staticness of the argument of f, to determine γ satisfying the *safety condition* $\gamma \circ f \sqsubseteq f \circ \delta$. For example, taking γ to be SND

satisfies $\gamma \circ \mathit{swap} \sqsubseteq \mathit{swap} \circ \mathit{FST}$ for *swap* defined by $\mathit{swap}\ (x, y) = (y, x)$. Taking γ to be BOT always satisfies the safety condition but tells nothing; greater γ is more informative. Launchbury [Lau91a] showed that this safety condition, in a sense which he formalises, is equivalent to the correctness condition for binding-time analysis in the general framework of Jones [Jon88]. Using projection-based analysis, Launchbury implemented both monomorphic and polymorphic versions of a partial evaluator for a first-order language.

There have been three notable attempts to generalise Launchbury's techniques to higher order. The first was Mogensen's generalisation of the polymorphic technique [Mog89]. Though successfully implemented, there is no formal statement of what it means for the analysis to be correct; even if such a statement were made, proving correctness would likely be difficult because of the highly intensional nature of the analysis: the non-standard values associated with expressions are strongly dependent on their syntactic structure, and projections are encoded symbolically as *abstract closures*, with approximation performed algebraically 'on-the-fly' according to time and space considerations. Nonetheless, the experiment provided evidence for the *practicality* of the projection-based approach at higher order.

The second generalisation was Hunt and Sands', of the monomorphic technique to higher order [HS91]. Their observation was that a projection, regarded as a set of domain-range pairs, is an *equivalence relation*: given γ, values u and v are in the same equivalence class if $\gamma\ u = \gamma\ v$, and the canonical elements of the equivalence classes are the set of fixed points (range) of γ. Hunt showed that the safety condition $\gamma \circ f \sqsubseteq f \circ \delta$ holds iff f is related to itself by $\delta \to \gamma$ where $\to$ is the standard operation on binary relations, so

$$(\gamma \circ f \sqsubseteq f \circ \delta) \Leftrightarrow (\forall\ u, v\ .\ \delta\ u = \delta\ v \ \Rightarrow\ \gamma\ (f\ \ u) = \gamma\ (f\ \ v))\ .$$

Then, for example, $(\mathit{BOT} \to \mathit{ID})\ (f, f)$ asserts that f maps dynamic arguments to static results. In general $\delta \to \gamma$ is not an equivalence relation, but it is always a *partial* equivalence relation: it is symmetric and transitive but not necessarily reflexive. (Partialness is crucial: if $\delta \to \gamma$ were in general reflexive then $(\delta \to \gamma)(f, f)$ would hold for all f, hence such assertions would tell nothing.) Unlike projections regarded as relations, PERs are closed under $\to$; the result, as Hunt and Sands show, is that 'scaling up' to higher-order analysis is reasonably straightforward. One disadvantage of their method is that PER spaces are considerably larger than the projection spaces on the same domains, and it is not clear which PERs to choose for (finite) abstract domains. Here they borrowed heavily from the projection world, using standard abstract projection domains at ground types. Further, its practicality has not been demonstrated by implementation, and, because of the unfamiliar territory, a promising route to a polymorphic generalisation is obscure.

The third generalisation was ours, to an entirely projection-based, monomorphic, higher-order technique [Dav93]. One observation motivating the approach is that there is no meaningful abstraction of values to projections, only of functions f to projection transformers τ (functions from projections to projections) satisfying $(\tau\ \delta) \circ f \sqsubseteq f \circ \delta$ for all δ. To make this abstraction possible a semantics intermediate between the standard and analysis semantics was introduced. Moving from the standard to intermediate semantics involved a translation of each ground type `T` to a function type `E -> T` (for a fixed type `E`), the values of which, being functions, could then be abstracted. The result, while proven correct with respect to a formal safety condition, is probably not practicable because of the growth in the sizes of (usefully rich) abstract domains induced by the type translation.

This paper presents a technique far simpler than our previous one. No intermediate semantics is required, and the correctness condition and proof are much simpler. Because the translation of ground types to function types is avoided the abstract domains are much smaller. Though entirely projection-based, we show that there is a reading of the results in terms of PERs, intimating a close relationship with the PER-based technique.

2 Language and Standard Semantics

The source language is a simple, strongly typed, monomorphic, non-strict functional language. The grammar for the language of types and type definitions is given in Figure 1. Nullary product corresponds to the so-called *unit*

$$
\begin{array}{lll}
\mathtt{T} ::= & \mathtt{A} & \text{[Type Name]} \\
\mid & \mathtt{Int} & \text{[Integer]} \\
\mid & (\mathtt{T}_1, \ldots, \mathtt{T}_n) & \text{[Product, } n \geq 0\text{]} \\
\mid & \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n & \text{[Sum, } n \geq 1\text{]} \\
\mid & \mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2 & \text{[Function]} \\
\mathtt{D} ::= & \mathtt{A}_1\ \mathtt{=}\ \mathtt{T}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{A}_n\ \mathtt{=}\ \mathtt{T}_n & \text{[Type Definitions]}
\end{array}
$$

Figure 1: Types and type definitions.

$$
\begin{array}{lll}
\mathtt{e} ::= & \mathtt{x} & \text{[Variable]} \\
\mid & \mathtt{n}_i & \text{[Numeral]} \\
\mid & \mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2 & \text{[Integer addition]} \\
\mid & \mathtt{e}_1\ \mathtt{e}_2 & \text{[Function application]} \\
\mid & (\mathtt{e}_1, \ldots, \mathtt{e}_n) & \text{[Tuple construction]} \\
\mid & \mathtt{let}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n)\ \mathtt{=}\ \mathtt{e}_0\ \mathtt{in}\ \mathtt{e}_1 & \text{[Tuple decomposition]} \\
\mid & \mathtt{c}_i\ \mathtt{e} & \text{[Sum construction]} \\
\mid & \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \{\mathtt{c}_i\ \mathtt{x}_i\ \mathtt{->}\ \mathtt{e}_i\} & \text{[Sum decomposition]} \\
\mid & \backslash\mathtt{x:T.e} & \text{[Lambda abstraction]} \\
\mid & \mathtt{e}_1\ \mathtt{e}_2 & \text{[Function application]} \\
\mid & \mathtt{fix}\ \mathtt{e} & \text{[Fixed point]}
\end{array}
$$

Figure 2: Expressions.

type. A unary product `(T)` will always have the same semantics as `T`. The types used in the examples are defined as follows.

```
FunList =   nil ()
          + cons (Int -> Int, FunList) ,

FunTree =   leaf (Int -> Int)
  + branch (FunTree, FunTree) .
```

The grammar for expressions is given in Figure 2. Addition for integers is provided as typical of operations on flat data types in this setting. A unary tuple `(e)` will always have the same semantics as `e`. The (monomorphic) typing of expressions is entirely standard and is omitted.

2.1 Expression semantics

Since two different expression semantics will be given, following Abramsky [Abr90] we define a semantics $\mathcal{E}$ parameterised by a set of *defining constants*. The semantics $\mathcal{E}$ is defined in Figure 3; the defining constants are *plus*, sel_i, *tuple*, inc_i, $outc_i$, *choose*, *mkfun*, *apply*, and *fix*. The two instances of $\mathcal{E}$ are distinguished by a superscript:

$$\mathcal{E}[\![\, \texttt{x}_i \,]\!]\; \rho \;=\; \rho[\![\, \texttt{x}_i \,]\!] \;=\; \mathit{sel}_i\; \rho$$

$$\mathcal{E}[\![\, \texttt{e}_1 \texttt{ + } \texttt{e}_2 \,]\!]\; \rho \;=\; \mathit{plus}\; (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\; \rho,\; \mathcal{E}[\![\, \texttt{e}_2 \,]\!]\; \rho)$$

$$\mathcal{E}[\![\, \texttt{(e}_1\texttt{,...,e}_n\texttt{)} \,]\!]\; \rho \;=\; \mathit{tuple}\; (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\; \rho,\; \ldots,\; \mathcal{E}[\![\, \texttt{e}_n \,]\!]\; \rho)$$

$$\begin{aligned} &\mathcal{E}[\![\, \texttt{let (x}_1\texttt{,...,x}_n\texttt{) = e}_0 \texttt{ in e}_1 \,]\!]\; \rho \\ &\quad = \; \mathcal{E}[\![\, \texttt{e}_1 \,]\!]\; \rho[\texttt{x}_i \mapsto \mathit{sel}_i\; (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\; \rho) \mid 1 \le i \le n] \end{aligned}$$

$$\mathcal{E}[\![\, \texttt{c}_i \texttt{ e} \,]\!]\; \rho \;=\; \mathit{inc}_i\; (\mathcal{E}[\![\, \texttt{e} \,]\!]\; \rho)$$

$$\begin{aligned} &\mathcal{E}[\![\, \texttt{case e}_0 \texttt{ of c}_1 \texttt{ x}_1 \texttt{ -> e}_1\texttt{; ...; c}_n \texttt{ x}_n \texttt{ -> e}_n \,]\!]\; \rho \\ &\quad = \; \mathit{choose}\; (\, \mathcal{E}[\![\, \texttt{e}_0 \,]\!]\; \rho, \\ &\qquad\qquad\qquad \mathcal{E}[\![\, \texttt{e}_1 \,]\!]\; \rho[\texttt{x}_1 \mapsto \mathit{outc}_1\; (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\; \rho)], \\ &\qquad\qquad\qquad\quad \vdots \\ &\qquad\qquad\qquad \mathcal{E}[\![\, \texttt{e}_n \,]\!]\; \rho[\texttt{x}_n \mapsto \mathit{outc}_n\; (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\; \rho)]) \end{aligned}$$

$$\mathcal{E}[\![\, \texttt{\textbackslash x.e} \,]\!]\; \rho \;=\; \mathit{mkfun}\; (\lambda x.\mathcal{E}[\![\, \texttt{e} \,]\!]\; \rho[\texttt{x} \mapsto x])$$

$$\mathcal{E}[\![\, \texttt{e}_1 \texttt{ e}_2 \,]\!]\; \rho \;=\; \mathit{apply}\; (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\; \rho)\; (\mathcal{E}[\![\, \texttt{e}_2 \,]\!]\; \rho)$$

$$\mathcal{E}[\![\, \texttt{fix e} \,]\!]\; \rho \;=\; (\mathit{fix} \circ \mathit{apply})\; (\mathcal{E}[\![\, \texttt{e} \,]\!]\; \rho)$$

Figure 3: Parameterised semantics.

$\mathcal{E}^{\mathrm{S}}$ for the standard semantics and $\mathcal{E}^{\mathrm{P}}$ for the non-standard semantics. The corresponding type semantics, and defining constants, have the same superscripts.

It is useful to regard the free-variable environment of each expression as having some tuple type $\texttt{(T}_1\texttt{,...,T}_n\texttt{)}$, and environment lookup as indexing (as in a categorical semantics, or De Bruijn indexing); variables are indexed implicitly or explicitly by their index in the free variable environment. Then for both versions of the semantics and all expressions $\texttt{e}$ of type $\texttt{T}$ with environment of type $\texttt{(T}_1\texttt{,...,T}_n\texttt{)}$,

$$\mathcal{E}[\![\, \texttt{e} \,]\!] \;\in\; \mathcal{T}[\![\, \texttt{(T}_1\texttt{,...,T}_n\texttt{)} \,]\!] \;\to\; \mathcal{T}[\![\, \texttt{T} \,]\!]\;.$$

Noting that $\rho[\![\, \texttt{x}_i \,]\!]$ is short for $\mathit{sel}_i\; \rho$, environment update $\rho[\texttt{x}_i \mapsto v]$ is defined by

$$\mathit{tuple}\; (\mathit{sel}_1\; \rho,\; \ldots,\; \mathit{sel}_{i-1}\; \rho,\; v,\; \mathit{sel}_{i+1}\; \rho,\; \ldots,\; \mathit{sel}_n\; \rho)\;.$$

2.2 Standard semantics

The standard S type and expression semantics are defined in Figure 4. Domain + is separated (lifted) sum. For more generality, products are unlifted; a unary sum-of-products gives a more usual lifted product. Function types give rise to lifted function spaces as in Abramsky's lazy lambda calculus [Abr89]. The important feature of this

$$\mathcal{T}^{\mathrm{S}}[\![\, \texttt{Int} \,]\!] \;=\; \mathit{Int} \;=\; \mathbf{Z}_\perp$$

$$\mathcal{T}^{\mathrm{S}}[\![\, \texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)} \,]\!] \;=\; \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_1 \,]\!] \;\times\; \ldots \;\times\; \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{T}^{\mathrm{S}}[\![\, \texttt{c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + } \texttt{c}_n\ \texttt{T}_n \,]\!] \;=\; \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_1 \,]\!] \;+\; \ldots \;+\; \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_1 \texttt{ -> } \texttt{T}_2 \,]\!] \;=\; (\mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_1 \,]\!] \;\rightarrow\; \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T}_2 \,]\!])_\perp$$

$$\mathit{plus}^{\mathrm{S}}\ (x, y) \;=\; x + y$$

$$\mathit{tuple}^{\mathrm{S}}\ (x_1, \ldots, x_n) \;=\; (x_1, \ldots, x_n)$$

$$\mathit{sel}_i^{\mathrm{S}}\ (x_1, \ldots, x_n) \;=\; x_i$$

$$\mathit{inc}_i^{\mathrm{S}} \;=\; \mathit{in}_i \circ \mathit{lift}$$

$$\mathit{outc}_i^{\mathrm{S}}\ x \;=\; \mathit{drop} \circ \mathit{out}_i$$

$$\begin{aligned} \mathit{choose}^{\mathrm{S}}\ (\perp,\quad\ x_1, \ldots, x_n) \;&=\; \perp \\ \mathit{choose}^{\mathrm{S}}\ (\mathit{in}_i\ v,\ x_1, \ldots, x_n) \;&=\; x_i \end{aligned}$$

$$\mathit{mkfun}^{\mathrm{S}} \;=\; \mathit{lift}$$

$$\mathit{apply}^{\mathrm{S}} \;=\; \mathit{drop}$$

$$\mathit{fix}^{\mathrm{S}} \;=\; \mathit{lfp} \quad \text{[Least fixed point]}$$

Figure 4: Standard type and expression semantics.

approach is the modelling of the fact that functions are, in effect, embedded in the simplest of lazy data structures, which either have no WHNF—have value $\perp$—or have a WHNF and contain an applicable function value—have value *lift* f for some f. (This can be made even more explicit by introducing a unary sum type, whence λ becomes a constructor [Dav94].) Recursive type definitions give rise to recursive domain specifications which have the usual least-fixed-point solutions.

3 Domain Factorisation

A key observation of our previous work [Dav93] is that since there is no concept of staticness of the body of a lambda expression, there is no point in having projections on function spaces. Hence values are factored into their evaluable or *data* parts, and their unevaluable but applicable *forward* parts. For example, for expressions of function type `T -> U`, with $(T \rightarrow U)_\perp = \mathcal{T}^{\mathrm{S}}[\![\, \texttt{T -> U} \,]\!]$, we distinguish two degrees of definedness—between values of the form *lift* f and $\perp$, corresponding to expressions that do and do not have a WHNF, respectively. These two degrees of definedness may be encoded by the two-point domain $\mathbf{1}_\perp = \{\perp, \mathit{lift}\ ()\}$—the *data domain* corresponding to type `T -> U`. There are two distinct projections on $\mathbf{1}_\perp$, namely ID and BOT: denoted functions are either

$$data_{\texttt{Int}} = id_{\texttt{Int}} ,$$

$$data_{(\texttt{T}_1,\ldots,\texttt{T}_n)} = data_{\texttt{T}_1} \times \ldots \times data_{\texttt{T}_n} ,$$

$$data_{\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n} = data_{\texttt{T}_1} + \ldots + data_{\texttt{T}_n} ,$$

$$data_{\texttt{T}_1\texttt{->T}_2} = (\lambda x.())_\perp .$$

$$for_{\texttt{Int}} = \lambda x.() ,$$

$$for_{(\texttt{T}_1,\ldots,\texttt{T}_n)} = for_{\texttt{T}_1} \times \ldots \times for_{\texttt{T}_n} ,$$

$$\begin{array}{lcl} for_{\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n}\ \perp & = & \perp \\ for_{\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n}\ (in_i\ v) & = & (\perp,\ldots,\perp,v,\perp,\ldots,\perp) \quad [v \text{ in } i^{th} \text{ position}] , \end{array}$$

$$for_{\texttt{T}_1\texttt{->T}_2} = drop .$$

$$unfac_{\texttt{Int}}\ (n,()) = n ,$$

$$\begin{array}{l} unfac_{(\texttt{T}_1,\ldots,\texttt{T}_n)}\ ((d_1,\ldots,d_n),\ (f_1,\ldots,f_n)) = (v_1,\ldots,v_n) \\ \quad \text{where} \\ \qquad v_i = unfac_{\texttt{T}_i}\ (d_i,f_i),\ 1 \le i \le n , \end{array}$$

$$\begin{array}{l} unfac_{\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n}\ ((d_1,\ldots,d_n),\ (f_1,\ldots,f_n)) = (v_1,\ldots,v_n) \\ \quad \text{where} \\ \qquad v_i = unfac_{\texttt{T}_i}\ (d_i,f_i),\ 1 \le i \le n , \end{array}$$

$$\begin{array}{l} unfac_{\texttt{T}_1\texttt{->T}_2}\ (\perp,f) = \perp, \\ unfac_{\texttt{T}_1\texttt{->T}_2}\ (lift\ (),\ f) = lift\, f . \end{array}$$

Figure 5: Factorisation and unfactorisation functions.

static or dynamic. Here the forward domain is $T \to U$, and the factorisation of $(T \to U)_\perp$ is $\mathbf{1}_\perp \times (T \to U)$. The factorisation function $fac_{\texttt{T->U}} \in (T \to U)_\perp \to (\mathbf{1}_\perp \times (T \to U))$ is an embedding defined by $fac_{\texttt{T->U}}\ \perp = (\perp,\perp)$, and $fac_{\texttt{T->U}}\ (lift\ f) = (lift\ (),\ f)$ for all f. Every embedding *determines* a corresponding projection; this follows from the relations $g \circ h \sqsubseteq id$ and $h \circ g = id$ that hold between an embedding g and its corresponding projection h. The importance of factorisation being an embedding is that no information is lost—values may be recovered by unfactoring. Here the projection is given by $unfac_{\texttt{T->U}}\ (\perp, f) = \perp$ and $unfac_{\texttt{T->U}}\ (lift\ (),\ f) = lift\ f$.

Generally, the data domain corresponding to type `T` is $\mathcal{D}[\![\,\texttt{T}\,]\!]$, where $\mathcal{D}$ is defined exactly like $\mathcal{T}^{\mathrm{S}}$ except that function spaces are replaced by the one-point domain $\mathbf{1}$, that is, $\mathcal{D}[\![\,\texttt{T}_1\ \texttt{->}\ \texttt{T}_2\,]\!] = \mathbf{1}_\perp$ for all $\texttt{T}_1$ and $\texttt{T}_2$. The function $data_{\texttt{T}}$ from values in $\mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}\,]\!]$ to their data parts in $\mathcal{D}[\![\,\texttt{T}\,]\!]$ is a projection: it is like the identity except that values from function spaces are mapped into $\mathbf{1}$. The projection $data_{\texttt{T}}$ is defined in terms of the structure of `T` as given in Figure 5. The definition of $data_{\texttt{T}_1\texttt{->T}_2}$ uses function lifting, defined by $f_\perp\ \perp = \perp$ and $f_\perp\ (lift\ x) = lift\ (f\ x)$. Recursive type definitions give rise to recursive function specifications with exactly one solution.

In the same style as $\mathcal{D}[\![\,\texttt{T}\,]\!]$ and $data_{\texttt{T}}$ we define $\mathcal{T}^{\mathrm{S}}_*[\![\,\texttt{T}\,]\!]$ to give the forward domain for `T`, and $for_{\texttt{T}}$ to be the function mapping values in $\mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}\,]\!]$ to their forward parts in $\mathcal{T}^{\mathrm{S}}_*[\![\,\texttt{T}\,]\!]$, as follows. Roughly, $\mathcal{T}^{\mathrm{S}}_*$ is like $\mathcal{T}^{\mathrm{S}}$ with all lifting removed and sum replaced by product, so

$$\mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{Int}\,]\!] \;=\; \mathbf{1}\;,$$

$$\mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)}\,]\!] \;=\; \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}_n\,]\!]\;,$$

$$\mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n\,]\!] \;=\; \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}_n\,]\!]\;,$$

$$\mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}_1\ \texttt{->}\ \texttt{T}_2\,]\!] \;=\; \mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}_1\,]\!] \;\to\; \mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}_2\,]\!]\;.$$

The mapping from standard values to their forward parts is defined in Figure 5. We write $\mathit{fac}_{\texttt{T}}$ for $\lambda x.(\mathit{data}_{\texttt{T}}\ x,\ \mathit{for}_{\texttt{T}}\ x)$. Then $\mathcal{D}[\![\,\texttt{T}\,]\!] \times \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}\,]\!]$ is a factorisation of $\mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}\,]\!]$, and

$$\mathit{fac}_{\texttt{T}} \;\in\; \mathcal{T}^{\mathrm{S}}[\![\,\texttt{T}\,]\!] \;\to\; (\mathcal{D}[\![\,\texttt{T}\,]\!] \;\times\; \mathcal{T}_*^{\mathrm{S}}[\![\,\texttt{T}\,]\!])$$

is an embedding which, as mentioned, determines the corresponding projection $\mathit{unfac}_{\texttt{T}}$ as defined in Figure 5.

We give some further examples. The factorisation of *Int* is $\mathit{Int} \times \mathbf{1}$; more generally, the factorisation of any domain D corresponding to a type not containing `->` is just $D \times \mathbf{1}$. For `FunList` the standard domain is

$$\mu X\ .\ \mathbf{1}_\perp + ((\mathit{Int} \to \mathit{Int})_\perp \times X)\;,$$

that is, lists of functions from integers to integers. In factored form the domain is

$$(\mu X\ .\ \mathbf{1}_\perp + (\mathbf{1}_\perp \times X)) \;\times\; (\mu Y\ .\ (\mathit{Int} \to \mathit{Int}) \times Y)\;,$$

that is, pairs consisting of a list of values from $\{\perp,\ \mathit{lift}\ \perp\}$, and an infinite tuple of functions from integers to integers. Then the standard value of the list `cons (\x.x-1, cons (\x.x-2, nil ()))` is

$$\mathit{incons}\ (\mathit{lift}\ (\lambda x.x - 1),\ \mathit{incons}\ (\mathit{lift}\ (\lambda x.x - 2),\ \mathit{innil}\ ()))\;,$$

the data value is

$$\mathit{incons}\ (\mathit{lift}\ (),\ \mathit{incons}\ (\mathit{lift}\ (),\ \mathit{innil}\ ()))\;,$$

and the forward value is $(\lambda x.x - 1,\ (\lambda x.x - 2,\ (\perp,\ (\perp,\ (\perp,\ \ldots)))))$.

4 Projection Semantics

Standard values v and v' are related by nonstandard value (α, κ) when their data parts are related by projection α, and their forward parts are *logically related* [Abr90] by κ. At each type `T` the relation is $\mathcal{R}[\![\,\texttt{T}\,]\!]\ ((\alpha, \kappa), \cdot, \cdot)$, defined by

$$\begin{array}{l} \mathcal{R}[\![\,\texttt{T}\,]\!]\ ((\alpha, \kappa), v, v') \;=\; (\alpha\ d = \alpha\ d') \;\wedge\; \mathcal{R}_*[\![\,\texttt{T}\,]\!]\ (\kappa, f, f') \\ \qquad\qquad\qquad\qquad \text{where} \\ \qquad\qquad\qquad\qquad\quad (d, f) \;=\; \mathit{fac}_{\texttt{T}}\ v \\ \qquad\qquad\qquad\qquad\quad (d', f') \;=\; \mathit{fac}_{\texttt{T}}\ v'\;, \end{array}$$

where $\mathcal{R}_*[\![\,\texttt{T}\,]\!]$ is defined by

$$\mathcal{R}_*[\![\,\texttt{Int}\,]\!]\ ((), (), ()) \;=\; \mathit{True}\;,$$

$$\mathcal{R}_*[\![\,\texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)}\,]\!] \;=\; \mathcal{R}_*[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; \mathcal{R}_*[\![\,\texttt{T}_n\,]\!]\;,$$

$$\mathcal{R}_*[\![\,\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n\,]\!] \;=\; \mathcal{R}_*[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; \mathcal{R}_*[\![\,\texttt{T}_n\,]\!]\;,$$

$$\mathcal{R}_*[\![\,\texttt{T}_1\ \texttt{->}\ \texttt{T}_2\,]\!] \;=\; \mathcal{R}[\![\,\texttt{T}_1\,]\!] \;\to\; \mathcal{R}[\![\,\texttt{T}_2\,]\!]\;.$$

$$plus^{\mathrm{P}}\ ((\alpha,()),(\beta,())) \;=\; \begin{cases} (ID,()), & \text{if } \alpha = ID \text{ and } \beta = ID \\ (BOT,()), & \text{otherwise} \end{cases}$$

$$tuple^{\mathrm{P}}\ ((\alpha_1,\kappa_1),\ldots,(\alpha_n,\kappa_n)) \;=\; ((\alpha_1 \times \ldots \times \alpha_n),(\kappa_1,\ldots,\kappa_n))$$

$$sel_i^{\mathrm{P}}\ ((\alpha_1 \times \ldots \times \alpha_n),(\kappa_1,\ldots,\kappa_n)) \;=\; (\alpha_i,\kappa_i)$$

$$inc_i^{\mathrm{P}}\ (\alpha,\kappa) \;=\; (C_i\ \alpha,\ (\top,\ldots,\top,\kappa,\top,\ldots,\top))$$

$$outc_i^{\mathrm{P}}\ (\alpha,(\kappa_1,\ldots,\kappa_n)) \;=\; (OUTC_i\ \alpha,\ \kappa_i)$$

$$choose^{\mathrm{P}}\ ((\alpha,\kappa),x_1,\ldots,x_n) \;=\; \begin{cases} \bot, & \text{if } \alpha \not\sqsupseteq \sqcap_{1\le i\le n}\ C_i\ BOT \\ x_1 \sqcap \ldots \sqcap x_n, & \text{otherwise} \end{cases}$$

$$mkfun^{\mathrm{P}}\ f \;=\; (ID,f)$$

$$apply^{\mathrm{P}}\ (\alpha,f) \;=\; \begin{cases} \bot, & \text{if } \alpha = BOT \\ f, & \text{if } \alpha = ID \end{cases}$$

$$fix^{\mathrm{P}} \;=\; gfp \quad \text{[Greatest fixed point]}$$

Figure 6: Projection semantics.

Here $\times$ and $\rightarrow$ are the standard operations on *ternary* relations: for relations R and S we have $(R \times S)((x,x'),(y,y'),(z,z'))$ iff $R(x,y,z)$ and $S(x',y',z')$, and $(R \rightarrow S)(f,g,h)$ iff for all x, y, and z such that $R(x,y,z)$ we have $S(f\ x,\ g\ y,\ h\ z)$. Here, recursive type definitions give recursive relation specifications, which have inclusive least-fixed-point solutions. (A relation is inclusive if, when it holds for each element of a directed set, and in particular an ascending chain, it also holds at the limit. Such relations are sometimes called *admissible* or *directed complete*. Some work is required to show inclusivity of these recursively-defined relations.) Thus the P type semantics $\mathcal{T}^{\mathrm{P}}$ must be

$$\mathcal{T}^{\mathrm{P}}[\![\,\mathtt{T}\,]\!] \;=\; |\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,| \;\times\; \mathcal{T}_*^{\mathrm{P}}[\![\,\mathtt{T}\,]\!]\ ,$$

where $|\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,|$ is the lattice of projections on domain $\mathcal{D}[\![\,\mathtt{T}\,]\!]$, and $\mathcal{T}_*^{\mathrm{P}}$ is defined exactly like $\mathcal{T}_*^{\mathrm{S}}$ with superscript P everywhere replacing superscript S.

The defining constants for the projection semantics $\mathcal{E}^{\mathrm{P}}$ are given in Figure 6. The following notation is used for specifying projections on the data domains. For sum type $\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n$ with data domain $D_1 + \ldots + D_n$, define $C_i\ \alpha$ to be $ID + \cdots + ID + \alpha + ID + \cdots + ID$ where α is the i^{th} summand, $OUTC_i\ (\gamma_1 + \cdots + \gamma_n)$ to be γ_i, and $OUTC_i\ BOT$ to be BOT.

The Central Result. For expression $\mathtt{e}$ of type $\mathtt{T}$ and free-variable environment type $\mathtt{E}$, the functions $\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}\,]\!]$, $\mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}\,]\!]$, and $\mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}\,]\!]$ are logically related by $\mathcal{R}$, that is

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \rightarrow \mathcal{R}[\![\,\mathtt{T}\,]\!])\ (\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}\,]\!],\ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}\,]\!],\ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}\,]\!])\ .$$

Further, $\mathcal{R}[\![\,\mathtt{T}\,]\!]\ ((\alpha,\kappa),\cdot,\cdot)$ is a PER for all α and κ.

The bulk of proof is omitted; it consists of showing that the defining constants are similarly related, and a simple induction on the structure of expressions showing that if the defining constants are logically related, then so are the semantic functions $\mathcal{E}^{\mathrm{P}}$, $\mathcal{E}^{\mathrm{S}}$, and $\mathcal{E}^{\mathrm{S}}$. By way of example we consider the expression form $\mathtt{e}_1$ + $\mathtt{e}_2$ and the relevant defining constant *plus*. To show that $plus^{\mathrm{S}}$ and $plus^{\mathrm{P}}$ are correctly related we need to show that

$$((\mathcal{R}[\![\,\mathtt{Int}\,]\!] \times \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \to \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \ (plus^{\mathrm{P}}, plus^{\mathrm{S}}, plus^{\mathrm{S}}) \ .$$

By the definitions of $\to$ and $\times$ on ternary relations, this is equivalent to

$$(\mathcal{R}[\![\,\mathtt{Int}\,]\!] \times \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \ (x, y, z) \ \Rightarrow \ \mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (plus^{\mathrm{P}} \ x, \ plus^{\mathrm{S}} \ y, \ plus^{\mathrm{S}} \ z) \ ,$$

for all x, y, and z. In turn, this is equivalent to

$$\begin{aligned} &\mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (x, y, z) \ \wedge \ \mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (x', y', z') \\ \Rightarrow \ &\mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (plus^{\mathrm{P}} \ (x, x'), \ plus^{\mathrm{S}} \ (y, y'), \ plus^{\mathrm{S}} \ (z, z')) \end{aligned}$$

for all x, x', y, y', z, and z'; equivalently, for all values y, y', z, and z' from the integer domain $\mathcal{T}^{\mathrm{S}}[\![\,\mathtt{Int}\,]\!]$ we have

$$(ID \ y = ID \ z) \ \wedge \ (ID \ y' = ID \ z') \ \Rightarrow \ ID \ (y + y') = ID \ (z + z') \ ,$$

which trivially holds, and for $\alpha_1 \neq ID$ or $\alpha_2 \neq ID$ that

$$(\alpha_1 \ y = \alpha_1 \ z) \ \wedge \ (\alpha_2 \ y' = \alpha_2 \ z') \ \Rightarrow \ BOT \ (y + y') = BOT \ (z + z') \ ,$$

which is also trivial. For the inductive case $\mathtt{e}_1$ + $\mathtt{e}_2$ with environment type `E` we need to show that if

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \ (\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_1\,]\!], \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1\,]\!], \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1\,]\!])$$

and

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \ (\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_2\,]\!], \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_2\,]\!], \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_2\,]\!])$$

then

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \ (\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!], \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!], \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!]) \ .$$

Suppose $\mathcal{R}[\![\,\mathtt{E}\,]\!] \ (\rho^{\mathrm{P}}, \rho^{\mathrm{S}}, \rho'^{\mathrm{S}})$. Then by the induction hypothesis we have that $\mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (\alpha_i, v_i, v'_i)$, where $(\alpha_i, ()) = \mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_i\,]\!] \ \rho^{\mathrm{P}}$, $(v_i, ()) = \mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_i\,]\!] \ \rho^{\mathrm{S}}$, and $(v'_i, ()) = \mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_i\,]\!] \ \rho'^{\mathrm{S}}$, for $i = 1, 2$. Now

$$\mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!] \ \rho^{\mathrm{P}}, \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!] \ \rho^{\mathrm{S}}, \ \mathcal{E}^{\mathrm{S}}[\![\,\mathtt{e}_1 + \mathtt{e}_2\,]\!] \ \rho'^{\mathrm{S}})$$

iff

$$\begin{aligned} \mathcal{R}[\![\,\mathtt{Int}\,]\!] \ (&plus^{\mathrm{P}} \ ((\alpha_1, ()), (\alpha_2, ())), \\ &plus^{\mathrm{S}} \ ((v_1, ()), (v_2, ())), \\ &plus^{\mathrm{S}} \ ((v'_1, ()), (v'_2, ()))) \ , \end{aligned}$$

which holds since $plus^{\mathrm{P}}$ and $plus^{\mathrm{P}}$, and their arguments, are correctly related.

5 Abstract Domains

At each type `T` we require a finite abstraction of $\mathcal{T}^{\mathrm{P}}[\![\,\mathtt{T}\,]\!]$. This abstract domain is $FProj_{\mathtt{T}} \times FFor_{\mathtt{T}}$, where $FProj_{\mathtt{T}}$ is a finite abstraction of the lattice $|\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,|$, and $FFor_{\mathtt{T}}$ is a finite abstraction of $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}\,]\!]$. The definition of $FProj_{\mathtt{T}}$ is based on Launchbury's [Lau91a]. A projection γ is in $FProj_{\mathtt{T}}$ if γ **proj** `T` can be inferred from the rules given in Figure 7. For recursively-defined types the rules yield only those projections that are *uniform*—that act on

$$BOT\ \ \mathbf{proj}\ \ \mathtt{Int} \qquad ID\ \ \mathbf{proj}\ \ \mathtt{Int}$$

$$BOT\ \ \mathbf{proj}\ \ \mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2 \qquad ID\ \ \mathbf{proj}\ \ \mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2$$

$$BOT\ \ \mathbf{proj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n$$

$$\frac{\gamma_1\ \ \mathbf{proj}\ \ \mathtt{T}_1\ \ \cdots\ \ \gamma_n\ \ \mathbf{proj}\ \ \mathtt{T}_n}{(C_1\ \gamma_1) \sqcap \cdots \sqcap (C_n\ \gamma_n)\ \ \mathbf{proj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}$$

$$\frac{\gamma_1\ \ \mathbf{proj}\ \ \mathtt{T}_1\ \ \cdots\ \ \gamma_n\ \ \mathbf{proj}\ \ \mathtt{T}_n}{\gamma_1 \times \ldots \times \gamma_n\ \ \mathbf{proj}\ \ (\mathtt{T}_1,\ldots,\mathtt{T}_n)}$$

$$\frac{\begin{array}{l}\gamma_1\ \ \mathbf{proj}\ \ \mathtt{A}_1\ \ \cdots\ \ \gamma_n\ \ \mathbf{proj}\ \ \mathtt{A}_n \\ \vdash\ P(\gamma_1,\ldots,\gamma_n)\ \ \mathbf{proj}\ \ \mathtt{T}_i(\mathtt{A}_1,\ldots,\mathtt{A}_n)\end{array}}{\mu(\gamma_1,\ldots,\gamma_n).P(\gamma_1,\ldots,\gamma_n)\ \ \mathbf{proj}\ \ \mathtt{A}_i} \qquad [\text{where } \mathtt{A}_i\mathtt{=}\mathtt{T}_i(\mathtt{A}_1,\ldots,\mathtt{A}_n)]$$

Figure 7: Inference rules for finite projection domains.

each recursive instance of a data structure in the same way. Thus $\mathit{FProj}_{\mathtt{FunList}}$ comprises BOT and $SPINE\ \alpha$ for α ranging over BOT and ID, where

$$SPINE\ \alpha \ = \ \mu\gamma.(NIL\ ID)\ \sqcap\ (CONS\,(\alpha\ \times\ \gamma))\ ,$$

so $SPINE\ BOT$ specifies static spines and dynamic elements, and $SPINE\ ID$ is the identity, thus specifying static spines and static elements. More generally, the abstract list constructor is isomorphic to lifting. Similarly, $\mathit{FProj}_{\mathtt{FunTree}}$ comprises BOT and $LBR\ \alpha$ for α ranging over BOT and ID, where

$$LBR\ \alpha \ = \ \mu\gamma.(LEAF\ \alpha)\ \sqcap\ (BRANCH\,(\gamma\ \times\ \gamma))\ ,$$

so $LBR\ \alpha$ specifies static branches and leaves, and α of the values in the leaf nodes. Again, the abstract tree constructor is isomorphic to lifting. This compares favourably with BHA strictness analysis, for which the corresponding abstract constructors are typically *double* lifting [Wad87, Sew93].

Value κ is in $\mathit{FFor}_{\mathtt{T}}$ if κ **fabsf** `T` can be inferred from the following. First, there is only one forward value at type `Int`.

$$()\ \ \mathbf{fabsf}\ \ \mathtt{Int}\ .$$

For products and sums,

$$\frac{\kappa_1\ \ \mathbf{fabsf}\ \ \mathtt{T}_1\ \ \cdots\ \ \kappa_n\ \ \mathbf{fabsf}\ \ \mathtt{T}_n}{(\kappa_1,\ldots,\kappa_n)\ \ \mathbf{fabsf}\ \ (\mathtt{T}_1,\ldots,\mathtt{T}_n)}\ ,$$

$$\frac{\kappa_1\ \ \mathbf{fabsf}\ \ \mathtt{T}_1\ \ \cdots\ \ \kappa_n\ \ \mathbf{fabsf}\ \ \mathtt{T}_n}{(\kappa_1,\ldots,\kappa_n)\ \ \mathbf{fabsf}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ .$$

Function spaces consist of a set of step functions closed under lub.

$$\frac{\begin{array}{cc}\alpha_1 \in \mathit{FProj}_{\mathtt{T}_1} & \kappa_1 \ \mathbf{fabsf}\ \mathtt{T}_1 \\ \alpha_2 \in \mathit{FProj}_{\mathtt{T}_2} & \kappa_2 \ \mathbf{fabsf}\ \mathtt{T}_2\end{array}}{\mathit{step}\ ((\alpha_1,\kappa_1),(\alpha_2,\kappa_2))\ \ \mathbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2)}\ ,$$

where

$$\begin{array}{rcl}\mathit{step}\ (v_1,v_2)\ x &=& v_2,\ \text{ if } v_1 \sqsubseteq x \\ \mathit{step}\ (v_1,v_2)\ x &=& \bot,\ \text{ otherwise },\end{array}$$

and

$$\frac{\kappa_1\ \ \mathbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2) \qquad \kappa_2\ \ \mathbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2)}{(\kappa_1 \sqcup \kappa_2)\ \ \mathbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2)}\ .$$

This gives the full space of monotonic functions on the abstract domains.

For recursively-defined types, roughly speaking, we choose those forward values that represent each component of the same type by the same value. Given type definitions $\mathtt{A}_1\ \mathtt{=}\ \mathtt{T}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{A}_n\ \mathtt{=}\ \mathtt{T}_n$, which will be written $\mathtt{A}_i\ \mathtt{=}\ \mathtt{T}_i\mathtt{(A}_1\mathtt{,...,A}_n\mathtt{)}$, $1 \le i \le n$, if from assumptions κ_i **fabsf** $\mathtt{A}_i$ for $1 \le i \le n$ it may be deduced that $P_i(\kappa_1,\ldots,\kappa_n)$ **fabsf** $\mathtt{T}_i(\mathtt{A}_1 \ldots \mathtt{A}_n)$ for $1 \le i \le n$, then

$$\mu(\kappa_1,\ldots,\kappa_n).(P_1(\kappa_1,\ldots,\kappa_n),\ldots,P_n(\kappa_1,\ldots,\kappa_n))$$

is a tuple $(\kappa_1,...,\kappa_n)$ such that κ_i **fabsf** $\mathtt{A}_i$ for $1 \le i \le n$. For all $\mathtt{T}$ the set $\mathit{FProj}_{\mathtt{T}} \times \mathit{FFor}_{\mathtt{T}}$ is a finite lattice containing the top and bottom elements of $\mathcal{T}^{\mathrm{P}}[\![\,\mathtt{T}\,]\!]$.

For `T` not containing `->` the domain $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}\,]\!]$ is isomorphic to $\mathbf{1}$, so $\mathit{FFor}_{\mathtt{Int}}$ is $\mathbf{1}$. For `Int -> Int` we have $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{Int\ ->\ Int}\,]\!] = (|\,\mathcal{D}[\![\,\mathtt{Int}\,]\!]\,| \times \mathbf{1}) \to (|\,\mathcal{D}[\![\,\mathtt{Int}\,]\!]\,| \times \mathbf{1})$, so $\mathit{FFor}_{\mathtt{Int->Int}}$ is $(\mathit{FProj}_{\mathtt{Int}} \times \mathbf{1}) \to (\mathit{FProj}_{\mathtt{Int}} \times \mathbf{1})$. A data structure of recursive type `A=T(A)` may be thought of as some (possibly infinite) number of elements of `T(())`. For example, the value $\mathit{cons}\ (f,\ \mathit{cons}\ (g,\ \mathit{nil}\ ()))$ in the standard domain for `FunList` decomposes into $\mathit{cons}\ (f,())$, $\mathit{cons}\ (g,())$, and $\mathit{nil}\ ()$. The (implicit) abstraction function maps such a data structure to the greatest lower bound of these elements, giving a safe abstraction of the nonstandard values of each element. Thus $\mathit{FFor}_{\mathtt{A=T(A)}} = \mathit{FFor}_{\mathtt{T(())}}$, so $\mathit{FFor}_{\mathtt{FunList}} \cong \mathit{FFor}_{\mathtt{FunTree}} \cong \mathit{FFor}_{\mathtt{Int->Int}}$.

6 Examples of Analysis

For all closed expressions `e` the abstract value $\mathcal{E}^{\mathrm{P}}[\![\,\mathtt{e}\,]\!]\,[\,]$ of `e` is of the form (ID,κ), showing that closed expressions are always entirely static. For expressions of function type the abstract forward value κ is a function from abstract arguments of `e` to abstract results.

First we consider functions on lists. Let expression `length` denote the usual length function:

```
fix (\length . \xs . case xs of
                        nil u  -> 0
                        cons p -> let (z,zs) = p in
                                       1 + length zs)
```

The abstract forward value of `length` is of the form $\lambda(\alpha,\kappa).(\tau\ \alpha,())$, where τ maps *SPINE BOT* and *SPINE ID* to *ID*, and *BOT* to *BOT*. This reveals that the result of `length` is independent of the values of list elements, and gives a static result when the argument has a static spine. It is worth pointing out that the result of `length` is static—determined—for infinite-length arguments with static spines; staticness and definedness are independent properties.

Let `append` stand for the expression denoting the usual function for appending two lists:

```
fix (\append . \xs . \ys . case xs of
                             nil u  -> ys
                             cons p -> let (z,zs) = p in
                         cons (z,
  append zs ys))
```

The abstract value of `append` is

$$\begin{array}{l}(ID,\ \lambda(\alpha_{\tt xs},\kappa_{\tt xs}).\\ (ID,\ \lambda(\alpha_{\tt ys},\kappa_{\tt ys}).\\ (\alpha_{\tt xs}\sqcap\alpha_{\tt ys},\ \kappa_{\tt xs}\sqcap\kappa_{\tt ys})))\ .\end{array}$$

This reveals that partial applications of `append` are static up to WHNF, and the abstract value of the result is the greatest lower bound of the two arguments. In general, the abstract value of a closed expression of the form $\backslash\mathtt{x}_1.\backslash\mathtt{x}_2\ldots.\mathtt{e}$ will reveal that all partial applications of the expression are static up to WHNF.

Let `reverse1` stand for the expression denoting the naive reverse function:

```
fix (\reverse1 . \xs . case xs of
                         nil u  -> nil ()
                         cons p -> let (z,zs) = p in
                                      append (reverse zs)
                    (cons (z, nil ())))
```

The abstract forward value of `reverse1` is the identity, so the abstraction of the elements of a list doesn't change by reversing the list.

Let `compose` stand for `\f.\g.\x.f (g x)`. The abstract value of `compose` is

$$\begin{array}{l}(ID,\ \lambda(\alpha_{\tt f},\kappa_{\tt f}).\\ (ID,\ \lambda(\alpha_{\tt g},\kappa_{\tt g}).\\ (ID,\ \left\{\begin{array}{ll}\bot, & \text{if } \alpha_{\tt f}=\mathit{BOT} \text{ or } \alpha_{\tt g}=\mathit{BOT}\\ \kappa_{\tt f}\circ\kappa_{\tt g}, & \text{otherwise}\end{array}\right\})))\ .\end{array}$$

Thus, the result of the application of the composition of two functions is dynamic, and maps all values to dynamic values, if either function is dynamic; otherwise, the result is given by the application of the composition of the abstract forward values to the abstract argument.

Let `listcomp` stand for the expression denoting the function that composes lists of functions:

```
fix (\listcomp . \fs . case fs of
                         nil u  -> \x.x
                         cons p -> let (g,gs) = p in
                                      compose g (listcomp gs))
```

The abstract value of `listcomp` is

$$(ID,\ \lambda(\alpha_{\tt fs},\kappa_{\tt fs}).(ID,\ \left\{\begin{array}{ll}\bot, & \text{if } \alpha_{\tt fs}\neq \mathit{SPINE\ ID}\\ \sqcap_{i\geq 0}(\kappa_{\tt fs})^i, & \text{otherwise}\end{array}\right\}))\ .$$

Since the abstract values of lists contain no information about the length of the lists of which they are abstractions, the abstract value of the composition of list elements is the glb of the composition over all lengths.

Let `flatten` stand for the expression denoting the function that flattens trees into lists:

```
fix (\flatten . \t . case t of
               leaf l   -> cons (l, nil ())
               branch p -> let (t1,t2) = p in
             append (flatten t1)
    (flatten t2))
```

The abstract value of `flatten` is

$$(ID,\ \lambda(\alpha_{\mathtt{fs}},\kappa_{\mathtt{fs}}).(\tau\ \alpha,\ \begin{Bmatrix} \bot, & \text{if } \tau\ \alpha_{\mathtt{fs}} \neq \mathit{LBR}\ \mathit{ID} \\ \kappa_{\mathtt{fs}}, & \text{otherwise} \end{Bmatrix}))\ ,$$

where τ maps BOT to BOT, and $LBR\ \alpha$ to $SPINE\ \alpha$ for α ranging over ID and BOT.

The function denoted by `compose listcomp flatten` that composes trees of functions has abstract value

$$(ID,\ \lambda(\alpha_{\mathtt{fs}},\kappa_{\mathtt{fs}}).(ID,\ \begin{Bmatrix} \bot, & \text{if } \alpha_{\mathtt{fs}} \neq \mathit{LBR}\ \mathit{ID} \\ \sqcap_{i\geq 0}(\kappa_{\mathtt{fs}})^i, & \text{otherwise} \end{Bmatrix}))\ .$$

Similarly to the case for lists, the abstract values of trees contain no information about the structure of the trees of which they are abstractions, so the abstract value of the composition of the values of the leaves is the glb of the composition over all tree structures.

7 More on Abstract Domains

The sizes of the abstract domains and the representations of the abstract values can be considerably optimised. The nonstandard semantics of application—embodied by $\mathit{apply}^{\mathrm{P}}$—guarantees that the abstract values (BOT, f) and (BOT, f') from $\mathcal{T}^{\mathrm{P}}[\![\,\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2\,]\!]$, for all f and f', are effectively the same: $\mathit{apply}^{\mathrm{P}}\ (BOT, f) = \bot$ for all f. A practical analyser would take advantage of this fact, identifying (BOT, f) over all f. More generally, for function types embedded within data structures, e.g. $(\mathtt{Int},\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2)$, abstract values $((\alpha,()),(BOT,f))$ would be identified over all f.

In the following we restrict attention to *denotable* values: at each type `T` those values that can be expressed as $\mathcal{E}[\![\,\mathtt{e}\,]\!]\ []$ for some `e`. For every value $v \in \mathcal{T}^{\mathrm{S}}[\![\,\mathtt{T}\,]\!]$ there is a best abstraction—a greatest value $(\alpha,\kappa) \in \mathcal{T}^{\mathrm{P}}[\![\,\mathtt{T}\,]\!]$ such that $\mathcal{R}[\![\,\mathtt{T}\,]\!]((\alpha,\kappa),v,v)$. For types $\mathtt{T}_1$ and $\mathtt{T}_2$ not containing `->` we have $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2\,]\!] \cong (|\,\mathcal{D}[\![\,\mathtt{T}_1\,]\!]\,| \times \mathbf{1}) \to (|\,\mathcal{D}[\![\,\mathtt{T}_2\,]\!]\,| \times \mathbf{1})$, and if $\kappa \in \mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}\,]\!]$ is greatest such that $\mathcal{R}_{*}[\![\,\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2\,]\!]\ (\kappa, f, f)$ for denotable f, then κ maps $(ID,())$ to $(ID,())$ and distributes over $\sqcap$. The subset of functions from $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2\,]\!]$ that map $(ID,())$ to $(ID,())$ and distribute over $\sqcap$ forms a complete lattice, hence attention may be restricted to this subset. Not only does this reduce the number of abstract functions, it also reduces their representation: such functions are determined by their behaviour on the unique $\sqcap$-*basis* of the lattice $|\,\mathcal{D}[\![\,\mathtt{T}_1\,]\!]\,|$—the set B of values such that every element of $|\,\mathcal{D}[\![\,\mathtt{T}_1\,]\!]\,|$ is the glb of some (possibly empty) subset of B, and no element of B can be expressed as the glb of some subset not containing that element. This optimisation can be generalised to higher order: $\mathcal{T}^{\mathrm{P}}_{*}[\![\,\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2\,]\!]$ may be restricted to functions that map $\top$ to $\top$ and distribute over $\sqcap$, for all function types $\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2$. (These results follow from those shown in earlier work [Dav94].)

8 Related Work

Consel [Con90] describes a binding-time analysis for higher-order untyped languages. As in our analysis abstract values have two parts, the first describing the static/dynamic properties of values, and the second describing how (for function types) abstract arguments are mapped to abstract results; there appears to be an implicit domain factorisation similar to ours, based on implicit type information collected by analysis semantics. In this respect there are many superficial similarities between the two techniques. No formal relation to the standard semantics is given, making a formal comparison with our technique problematic.

Hughes [Hug88] proposes a general approach to promoting related analysis techniques to higher order; instantiation to projection-based analysis yields a technique much more closely related to that of our previous work [Dav93] than to the one given here: transformation of his equations to the same general form as ours reveals that translation of ground types to function types (as previously mentioned) is implicit in the technique.

9 Conclusion

We have successfully generalised Launchbury's polymorphic partial-projection-based demand analysis to higher-order, using abstract domains smaller than those typically used in BHA strictness analysis.

References

[Abr89] S. Abramsky. "The lazy lambda calculus." *Research Topics in Functional Programming.* David Turner, ed., Addison-Wesley 1989.

[Abr90] S. Abramsky. "Abstract interpretation, logical relations and Kan extensions." *Journal of Logic and Computation*, 1, 1990.

[Bar91] G. Baraki. "A note on abstract interpretation of polymorphic functions." In [Hug91].

[Bar93] G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions.* Ph.D. thesis, Research report FP-1993-7, Department of Computing Science, University of Glasgow.

[BEJ88] D. Bjorner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Con90] C. Consel. "Binding Time Analysis for Higher Order Untyped Functional Languages." Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp264-272.

[Dav93] K. Davis. "Higher-order Binding-time Analysis." *Proceedings of the 1993 ACM on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[Dav94] K. Davis. *Projection-based Program Analysis.* Ph.D. thesis, Computing Science Department, The University of Glasgow, 1994.

[Go92] C.K. Gomard. "A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics." ACM TOPLAS, Vol 14, No. 2, April 1992.

[HM94] F. Henglein and C. Mossin. "Polymorphic binding-time analysis." European Symposium on Programming (ESOP '94).

[Hug88] J. Hughes. Backwards analysis of functional programs. In [BEJ88].

[Hug91] J. Hughes, ed. *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, Cambridge, Sept 1991. LNCS 523, Springer Verlag, 1991.

[Hun91] S. Hunt. "PERs generalise projections for strictness analysis (extended abstract)." *Proceedings of the 1990 Glasgow Workshop on Functional Programming.* Simon L. Peyton Jones *et al.*, eds. Springer Workshops in Computing. Springer-Verlag, 1991.

[HS91] S. Hunt and D. Sands. "Binding time analysis: a new PERspective." *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices Vol. 26, No. 9, 1991.

[Jen92] T. Jensen. *Abstract Interpretation in Logical Form.* Ph.D. thesis, Report 93/11, Department of Computer Science, University of Copenhagen, 1992.

[Jon88] N.D. Jones. "Automatic program specialization: A re-examination from basic principles." In [BEJ88].

[Lau88] J. Launchbury. "Projections for specialisation." In [BEJ88].

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD Thesis, Glasgow University, Nov 89. Distinguished Dissertation in Computer Science, Vol 1, CUP, 1991.

[Mog89] T. Mogensen. "Binding-time analysis for polymorphically typed higher order languages." *International Joint Conference on Theory and Practice of Software Development.* LNCS 352. Springer-Verlag 1989.

[NN88] H.R. Nielson and F. Nielson. "Automatic binding-time analysis for a typed λ-calculus." *Science of Computer Programming 10*, North Holland, 1988. Also in POPL '88.

[Sch88] D.A. Schmidt. "Static properties of partial reduction." In [BEJ88].

[Sew93] J. Seward. "Polymorphic strictness analysis using frontiers." *Proceedings of the 1993 ACM on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[Wad87] P. Wadler. "Strictness analysis on non-flat domains by abstract interpretation over finite domains. " S. Abramsky, C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.