Higher Order Demand Propagation

Dirk Pape
Department of Computer Science
Freie Universität Berlin
pape@inf.fu-berlin.de

Abstract. A new denotational semantics is introduced for realistic non-strict functional languages, which have a polymorphic type system and support higher order functions and user definable algebraic data types. It maps each function definition to a demand propagator, which is a higher order function, that propagates context demands to function arguments. The relation of this "higher order demand propagation semantics" to the standard semantics is explained and it is used to define a backward strictness analysis. The strictness information deduced by this analysis is very accurate, because demands can actually be constructed during the analysis. These demands conform better to the analysed functions than abstract values, which are constructed alone with respect to types like in other existing strictness analyses. The richness of the semantic domains of higher order demand propagation makes it possible to express generalised strictness information for higher order functions even across module boundaries.

1 Introduction

Strictness analysis is one of the major techniques used in optimising compilers for lazy functional programs [15]. If a function is identified to be strict in some arguments, these arguments can safely (without changing the lazy semantics) be calculated prior to the function call, saving the effort of building a suspension and later entering it. Strictness information can also be used to identify concurrent tasks in a parallel implementation, since referential transparency guarantees, that function arguments can be reduced independently. *Generalised* strictness analysis can in addition derive an amount of evaluation, which is safe for a larger data structure (e.g. evaluating the spine of a list argument). Such information can be used to find larger tasks in a parallel implementation [2,7], hence reducing granularity. Though it is not always recommended to use generalised strictness information for a sequential implementation – because of the danger of introducing space leaks – it can in general improve simple strictness analysis results even in the sequential setting.

In this paper a new approach to generalised strictness analysis is proposed. The analysis is called demand propagation because evaluation demands are propagated from composed expressions to its components (*backward* w.r.t. function application). In contrast to other backward analyses like projection analysis [19,3] or abstract demand propagation [17] this analysis is applicable for a higher order *and* polymorphic language and works with infinite domains in the non-standard semantics.

In Sect. 2 a simple but realistic functional language based on the polymorphically typed lambda calculus is introduced together with its standard semantics.

K. Hammond, T. Davie, and C. Clack (Eds.): IFL'98, LNCS 1595, pp. 153-168, 1999.

Demands, demand propagators and the demand propagation semantics are defined in Sect. 3. The utilisation of demand propagators for generalised strictness analysis is explained in Sect. 4. The soundness proof for the semantics can be read in [13].

Higher order demand propagation departs from other strictness analyses: its abstract values are higher order functions, which calculate the generalised strictness information via backward propagation of demands. A serious implication of this is that the abstract domains are in general infinite. Because of this, more accurate strictness information of a function can be expressed and made available to functions which use it in their definition, even across module boundaries. This can be seen by looking at the examples in the appendix and in [12]. How the analysis can deal with those infinite domains is briefly sketched in Sect. 5 and more thoroughly in [12]. As a bonus it is possible to trade accuracy against speed, making the analysis applicable for different stages of program development.

Further research topics and related work are discussed in Sects. 6 and 7.

2 Core Language Syntax and Semantics

The simple but realistic functional language defined in this section is basedon the polymorphically typed lambda calculus. The language is higher order and provides user definable algebraic data types. It is a module language and the standard semantics of a module is a transformation of environments. The module's semantics transforms a given *import* environment into an *export* environment by adding to it the semantics of the functions, which are defined in the module.

2.1 Syntax of the Core Language

The syntax of the core language and its associated type language is given by:

type :== [∀ { **tvar** } **.**] *monotype*
monotype :== **tvar** | **INT** | *monotype* → *monotype* | **tcons** { *monotype* }
usertype :== [∀ { **tvar** } **.**] *algebraic*
algebraic :== [*algebraic* **+**] **cons** { *monotype* }
expr :== **var** | **cons** | *expr expr* | λ **var .** *expression* | **let** *module* **in** *expr*
| **case** *expr* **of** { **cons** { **var** } ⇒ *expr* } **var** ⇒ *expr*
module :== { **tcons =** *usertype* } { **var =** *expr* }

It consists of user type declarations and function declarations. Expressions can be built of variables, constructors, function applications, lambda abstractions, case- and let-expressions. All expressions are assumed to be typed, but for simplicity all type annotations are omitted in this paper. In a real implementation the principal types can be inferred by Hindley-Milner type inference. Nevertheless we define a type language which is used to index the semantic domains.

Polymorphic types are defined by universal quantification of all free type variables at the outermost level of the type, hence an expression's type must not contain a free type variable. The same holds for the user defined types. This is a common approach, that is e.g. followed in the definition of the functional language Haskell [14]. Constructors of an algebraic data type are assumed to be unique over all types.

The usual constants 0, 1, -1, …, +, *, … appear in the language as variables with their semantics assigned to them in a prelude environment. Integer numbers also represent constructors, which identify the components of the infinite sum $Z_\perp \cong (\{0\} \oplus \{1\} \oplus \{-1\} \oplus \ldots)_\perp$ and which can be used in a pattern of a case-alternative.

A case-expression must always contain a default alternative, to eliminate the necessity to handle pattern match errors in the semantics. An ordinary case-expression without a default alternative can easily be transformed by adding one with the expression `bot`, where `bot` is defined for all types and has the semantics $\perp$.

2.2 Standard Semantics of Core Language Modules

The definition of the standard semantics is given in Fig. 1 and consists of four parts:

1. **The Type Semantics $\mathcal{D}$:** The semantic domains associated to the types of the type language are complete partial ordered sets (cpos). They are constructed out of the flat domain for INT by sum-of-products construction for algebraic data types and continuous function space construction for functional types. These constructions yield again cpos as shown e.g. in [5]. Function domains are lifted here to contain a least element $\perp$ for functional expressions, which do not have a weak head normal form. A domain for a polymorphic type is defined as a type indexed family of domains. User type declarations can be polymorphic and mutual recursive. User defined types are represented by type constructors, which can be applied to the appropriate number of types yielding the domains for recursive data types, which themselves are defined as fixpoints of domain equations in the usual way [5]. Because we focus on the expression semantics here, we handle the type constructors as if they are predefined in the type environment for the type semantics.
2. **The Expression Semantics $\mathcal{E}$:** Syntactic expressions of a type τ are mapped to functions from an environment – describing the bindings of free variables to semantic values – into the domain belonging to τ. The semantics of polymorphic expressions are families of values for all instances of the polymorphic type.
3. **The User Type Declaration Semantics $\mathcal{U}$:** The sum-of-products domains for user defined data types come together with unique continuous injection functions in_c into the sum and with continuous projection functions $proj_{c,i}$ to the i-th factor of the summand corresponding to `C`. The user type definition semantics implicitly defines a constructor function for each constructor with the corresponding injection function as its semantics. For notational convenience we also define a function *tag*, which maps any non-bottom value of a sum to the constructor of the summand it belongs to. For $v \neq \perp$ it holds $v \in Image(in_{tag(v)})$.
4. **The Module Semantics $\mathcal{M}$:** The semantics of a core language module is a transformation of environments. Declarations in a core language module can refer to the semantics defined in the import environment. The semantics of the set of mutually recursive declarations in a module for a given import environment is the (always existing) minimal fixpoint of an equation describing the environment enrichment.

Fig. 1. Standard Semantics of the Core Language (cont.)

$\mathcal{E}: \forall\tau:\mathsf{Type}.\ \mathsf{Expr}^{\tau} \to \mathsf{Env} \to \mathcal{D}[[\tau]]$ where $\mathsf{Env} = \forall\tau.\ \mathsf{Var}^{\tau} \oplus \mathsf{Const}^{\tau} \to \mathcal{D}[[\tau]]$

$\mathcal{E}[[x]]\ \rho = \rho\ x$ þþþand $\mathcal{E}[[C]]\ \rho = \rho\ C$

$\mathcal{E}[[e_1\ e_2]]\ \rho = \mathcal{E}[[e_1]]\ \rho\ (\mathcal{E}[[e_2]]\ \rho)$; resp. $\perp$, if $\mathcal{E}[[e_1]]\ \rho = \perp$

$\mathcal{E}[[\lambda x.\ e]]\ \rho = \lambda v.\ \mathcal{E}[[e]]\ \rho[v/x]$

$\mathcal{E}[[\textbf{case}\ e\ \textbf{of}\ C_1\ v_{11}\ldots v_{1a1} \Rightarrow e_1;\ \ldots;\ C_n\ v_{n1}\ldots v_{nan} \Rightarrow e_n;\ v \Rightarrow e_{def}]]\ \rho$

$= \perp$, if $e = \perp$

$= \mathcal{E}[[e_i]]\ \rho[proj_{Ci,1}\ (e)/v_{i1},\ \ldots,\ proj_{Ci,ai}\ (e)/v_{iai}]$, if $tag(e)=C_i$

$= \mathcal{E}[[e_{def}]]\ \rho[e/v]$, otherwise

where $e = \mathcal{E}[[e]]\ \rho$

$\mathcal{E}[[\textbf{let}\ m\ \textbf{in}\ e]]\ \rho = \mathcal{E}[[e]]\ (\mathcal{M}[[m]]\ \rho)$

$\mathcal{U}: \mathsf{Usertype} \to \mathsf{Env} \to \mathsf{Env}$

$\mathcal{U}[[\forall\alpha_1\ldots\alpha_m.\ C_1\ \tau_{11}\ldots\tau_{1a1} + \ldots + C_n\ \tau_{n1}\ldots\tau_{nan}]] = [c_1/C_1,\ \ldots,\ c_n/C_n]$

where $c_i = \lambda v_1\ldots v_{ai}.\ in_{ci}\ (v_1,\ldots,v_{ai})$

$\mathcal{M}: \mathsf{Module} \to \mathsf{Env} \to \mathsf{Env}$

$\mathcal{M}[[T_1 = \upsilon_1;\ \ldots;\ T_m = \upsilon_m;\ fdefs]]\ \rho = \mathcal{F}[[fdefs]]\ (\rho ++ \mathcal{U}[[\upsilon_{1]]} ++ \ldots ++ \mathcal{U}[[\upsilon_{n]]})$

where $\mathcal{F}[[v_1 = e_1;\ \ldots;\ v_n = e_n]]\ \rho = \mu P.\ \rho[\mathcal{E}[[e_1]]\ P/v_1,\ \ldots,\ \mathcal{E}[[e_n]]\ P/v_n]$

$f = \mathcal{M}[[m]]\ \rho$ f is the semantics of a variable f in a module m with import env. ρ

Prelude semantics are provided for integer numbers, + and bot

3 Higher Order Demand Propagation Semantics

The aim of higher order demand propagation semantics is to express how evaluation demands on a given function application propagate to evaluation demands on the function arguments (lazy evaluation is presumed). The definition follows the structure of the definition of the standard semantics but maps the syntactical function definitions to so called demand propagators, which are defined below.

3.1 The Abstract Semantic Domains for Higher Order Demand Propagation

We now define the notions of demand and demand propagator. Note that demands and demand propagators are polymorphically typed. This is the reason why higher order demand propagation works for a polymorphic language.

Definition 1 (Demand). A *demand* Δ of type τ is a continuous function from the standard semantic domain $\mathcal{D}[[\tau]]$ to the two-point-cpos $2=\{1\}_{\perp}$ with $\perp<1$. The continuous function space of demands is again a cpos with $\Delta_1<\Delta_2$, iff $\Delta_1 v < \Delta_2 v$ for all $v \in \mathcal{D}[[\tau]]$.

The continuity of Δ implies the closedness of the set $\Delta^{-1}(\{\bot\})$ with respect to the Scott-topology of the cpos. And the characteristic function Δ_C on a Scott-closed set C with $\Delta_C(C) = \bot$ and $\Delta_C(C) = 1$ is continuous, which shows that the functional notion is equivalent to the usual notion of demands as Scott-closed subsets of the value domain. We prefer the functional notion because it provides an easy way to define demands on polymorphic domains namely by polymorphic characteristic functions.

Operationally, demands represent evaluation strategies like evaluators in [2]. A semantic value is mapped to $\bot$, if and only if the related evaluation strategy fails for that value.

We introduce three basic demands, which are fully polymorphic and can be applied to all values and which are defined by[1]:

NO $v = 1$, for all v (no evaluation)
WHNF $v = \bot$, iff $v = \bot$ (evaluation to weak head normal form)
FAIL $v = \bot$, for all v (non terminating evaluation).

Algebraic demands (e.g. for lists) can be constructed out of component demands:

(C $\Delta_1 \ldots \Delta_n$) v=inf $\{\Delta_1 v_1, \ldots, \Delta_n v_n\}$, if $tag(v)$=C, hence $v=in_c(v_1,\ldots,v_n)$; $=\bot$, otherwise.

Note that the evaluation strategy for a constructor demand – e.g. CONS WHNF NO – forces evaluation to a `Cons`-node and the evaluation of the head of that node to weak head normal form. If applied to an empty list `Nil`, it fails. Such constructor demands arise from the analysis of a case-alternative.

Definition 2 (Demand Operations). Demands can be combined by the operators | and &, which are defined as pointwise supremum resp. infimum. Algebraic demands can be *projected* to a component ($\downarrow$) or *restricted* to some summands of the sum (\). The projected demand is the demand induced on a specific factor of the sum-of-products. It arises from analysing constructor applications. The restricted demand is modified to be $\bot$ on all elements of a specified set of summands. Restriction is used to describe the propagation in a default alternative of a case-expression.

$(\Delta_1 \,\&\, \Delta_2)\ v = \inf\{\Delta_1 v, \Delta_2 v\}$ and $(\Delta_1 \mid \Delta_2)\ v = \sup\{\Delta_1 v, \Delta_2 v\}$

$\Delta\downarrow_{C,i}\ v = \bot$, iff for all $v_1 \ldots v_n$: $\Delta\,(in_c\,(v_1,\ldots,v,\ldots,v_n)) = \bot$ (v at i-th position)

$\Delta\backslash$CS $v = \bot$, if $tag(v) \in$ CS; $= \Delta\ v$, otherwise

Recursive demands can be defined as minimal fixpoints on the function space of demands, for example:

SPINE = $\mu\Delta$. NIL | CONS NO Δ (we also write: SPINE = NIL | CONS NO SPINE)

(EVEN,ODD) = $\mu(\Delta_1,\Delta_2)$. (NIL | CONS NO Δ_2 , NIL | CONS WHNF Δ_1)

Remark 1. The general demands NO, WHNF and FAIL are also defined for function types. But demand propagation semantics is not interested in more complex functional

[1]Our thanks to one referee who remarked that the definition of WHNF is not parametric polymorphic. We consider WHNF as an overloaded name for instances in every type domain.

demands which can be imagined, since only non-functional context demands (which do not have "→" as the outermost type constructor) are propagated through the demand propagators. The demand, which will be propagated by a demand propagator, is always interpreted as a demand on the non-functional result type of the function, which can be deduced from the function's type.

This corresponds to the common operational semantics of lazy functional languages, which do not evaluate function applications until they are satisfied, meaning they are provided with the number of arguments stated in their definition. Paradoxically the lack of functional demands seems to be crucial to make the demand propagation semantics higher order.

Definition 3 (More-Effective Relation). The induced cpo on the function space of demands, has FAIL as its bottom element and NO as a universal greatest element. Since it is counter-intuitive to say, that NO is the greatest demand, we define a new partial order on demands: A demand Δ is called *more effective* than Δ', noted $\Delta » \Delta'$, if and only if $\Delta < \Delta'$ with respect to the induced cpo. Δ is called *effective*, if and only if $\Delta » \mathrm{NO}$.

Hence WHNF is the least effective demand. All demands are comparable with NO and WHNF. And they are comparable with FAIL, the most effective demand.

Example 1 (Motivation of Demand Propagators). The definition of a demand propagator has to reflect that context demands are always meant as demands on the non-functional result type of an expression (see Remark 1), hence the type of the context demand is the same for all partial applications of a function. In general the demand which is propagated to an argument will depend on *all* arguments of the function in at least two senses:

1. It depends on the existence of arguments, since a function application will only be evaluated if it has a sufficient number of arguments.
2. If one argument is itself a function – which is a common case in higher order functional languages – and if this function is applied to the argument, to which the demand is to be propagated, the propagation may depend on the propagation of some demand through this argument function. This is e.g. the case if the strictness of the higher order function `map` in its second argument is analysed for a strict or for a non-strict function as the first argument (cf. Example 5 in the appendix).

To deal with this situation, the demand propagator of a function only describes the dependence of the propagation from the existence and the values of argument demand propagators. A simple example illustrates, how demand propagation works and introduces a simple notation for parameterised demands, the $\underline{\lambda}$-abstraction.

Let `cond` be the conditional function defined by

```
> cond = λb.λx.λy. case b of True ⇒ x; False ⇒ y
```

`cond` has the type $\forall\alpha$. Bool$\to\alpha\to\alpha\to\alpha$, where Bool = True + False is defined as a user type. TRUE and FALSE are the corresponding algebraic constructor demands. The demand propagator $\underline{\mathrm{COND}}$ for `cond` is noted by the following equation:

$$\underline{\mathrm{COND}}\ \Delta = \underline{\lambda}\mathrm{B}.\ \underline{\lambda}\mathrm{X}.\ \underline{\lambda}\mathrm{Y}.\ \underline{\mathrm{B}}\ \mathrm{WHNF}\ \&\ (\underline{\mathrm{B}}\ \mathrm{TRUE}\ \&\ \underline{\mathrm{X}}\ \Delta \mid \underline{\mathrm{B}}\ \mathrm{FALSE}\ \&\ \underline{\mathrm{Y}}\ \Delta)\ . \qquad (1)$$

COND is a function which takes a demand Δ as the first argument and yields a so called parameterised demand, noted by λ-abstractions. A λ-abstraction can be interpreted as follows: if it is applied to an argument demand propagator, then it is an ordinary lambda abstraction for a demand propagator, which may be free in the body. If it is *not* applied it can be read as the propagated demand NO, stating that no demand is propagated, because the function is not provided with a sufficient number of arguments. To achieve this dual semantics the λ-abstraction will later be defined as a pair of a propagated demand and an ordinary abstraction.

In our backward generalised strictness analysis we provide a context demand for a function application and aim to infer demands, which can be applied to the function's arguments without changing the lazy semantics. A demand propagator which does this job is called *safe* for this function (cf. Definition 6). It is obvious that a safe propagator can always propagate FAIL to FAIL and must propagate NO to NO, regardless of its argument propagators. Hence from now on we assume that all further demand propagator equations include these two cases and only note the case for effective non-failure demands. Thus with all these explanations in mind (1) can be unfolded to:

COND NO = (NO , λB. (NO , λX. (NO , λY. NO)))
COND FAIL = (FAIL , λB. (FAIL , λX. (FAIL , λY. FAIL)))
COND Δ = (NO , λB.(NO , λX.(NO , λY.B WHNF & (B TRUE & X Δ | B FALSE & Y Δ)))) , otherwise.

If COND is applied to an effective non-failure demand Δ and further to three argument demand propagators B and X and Y, the body B WHNF & (B TRUE & X Δ | B FALSE & Y Δ) of the abstraction states, that in any case a WHNF demand is propagated to the first argument, and either TRUE is propagated to the first argument and Δ to the second or FALSE to the first and Δ to the third argument.

The propagated demand now depends on the values provided for the arguments of the parameterised demand. When testing for generalised strictness COND Δ is applied to a combination of the special demand propagators NO and ID, specifying propagation by arbitrary arguments (propagating every context safely to NO) or by an argument being tested (reproducing the context demand). The following can be deduced:

COND Δ ID NO NO = ID WHNF & (ID TRUE & NO Δ | ID FALSE & NO Δ) = WHNF
COND Δ NO ID NO = NO WHNF & (NO TRUE & ID Δ | NO FALSE & NO Δ) = NO
COND Δ NO NO ID = NO WHNF & (NO TRUE & NO Δ | NO FALSE & ID Δ) = NO .

Hence `cond` is found to be strict in the first argument. If an application of `cond` to some statically known arguments is analysed in another function definition using `cond`, the argument propagators are statically known and may express the propagation to a shared subexpression (cf. Examples 2 and 3 in the appendix).

From these motivation we collect the following formal definitions:

Definition 4 (Demand Propagator, Parameterised Demand). A *demand propagator* for a function of type $\tau=\tau_1\to\ldots\to\tau_n\to\tau'$ (τ' non-functional), is a continuous function, which maps a demand of type τ' (the *context demand*) to a parameterised demand. A *parameterised demand* is a pair of the *propagated demand* and a continuous function, which maps an argument demand propagator to a new parameterised demand.

The domains are defined by the following mutual recursive equations:
$\mathsf{PDemand}^{\tau'} = \forall\alpha.\ \mathsf{Demand}^{\alpha}$
$\mathsf{PDemand}^{\tau} = \forall\alpha.\ \mathsf{Demand}^{\alpha} \times [\mathsf{Propagator}^{\tau 1}\ \alpha \to \mathsf{PDemand}^{\tau 2 \to \ldots \to \tau n \to \tau'}\ \alpha]$
$\mathsf{Propagator}^{\tau} = \forall\alpha.\ [\mathsf{Demand}^{\tau'} \to \mathsf{PDemand}^{\tau}\ \alpha]$.

A demand Δ can be lifted into a parameterised demand of any type τ by the functions lift^{τ} with $\mathit{lift}^{\tau}(\Delta) = \Delta$, if τ is not functional, and $\mathit{lift}^{\tau \to \pi'}(\Delta) = (\Delta, \mathit{lift}^{\pi'}(\Delta))$.

Definition 5 ($\underline{\lambda}$-Abstraction). Let $\pi = \pi(\underline{P})$ be a parameterised demand, which can depend on a demand propagator $\underline{P}$. Then $\underline{\lambda P}.\ \pi$ denotes the parameterised demand $(\mathsf{NO}, \lambda \underline{P}.\ \pi)$.

The following demand propagators are essential for general strictness analysis:
$\underline{\mathrm{NO}}^{\tau}\ \Delta = \mathit{lift}^{\tau}(\mathsf{NO})$, $\underline{\mathrm{STRICT}}^{\tau}\ \Delta = \mathit{lift}^{\tau}(\mathsf{WHNF})$ and
$\underline{\mathrm{ID}}^{\tau}\ \Delta = \Delta$, if τ is non-functional, and $\underline{\mathrm{ID}}^{\tau \to \pi'}\ \Delta = \underline{\mathrm{STRICT}}^{\tau \to \pi'\ \Delta}$

Remark 2. The definition of a parameterised demand for a function type as a pair reflects the fact that there are two things we want to do with it. The first purpose is to apply it to a known demand propagator for an argument of the function. This means that we want to apply its second component. On the other hand we want to see the parameterised demand as a propagated demand, for instance to propagate it further. In this case we refer to its first component. It would be a mess of notation, if we always wanted to write the projection to the correct component in a given utilisation of a parameterised demand. Fortunately from the context it is always clear which component is used. So the notation can be shortened in both cases. For example:

"$\underline{P}\ \Delta\ \underline{P}_1\ \underline{P}_2$" is the short form for "snd (snd ($\underline{P}\ \Delta$) $\underline{P}_1$) $\underline{P}_2$"

"$\underline{P}\ (\underline{P}_1\ \Delta)\ \underline{P}_2$" is the short form for "snd ($\underline{P}$ (fst ($\underline{P}_1\ \Delta$))) $\underline{P}_2$"

Only with this meaning it makes sense to say, a parameterised demand is applied to an argument demand propagator or is propagated by a demand propagator.

Remark 3. The operations on demands, which were introduced in Definition 2, generalise in a natural way to parameterised demands by applying them to the first component and recursively to the second (cf. [13]).

3.2 Definition of the Demand Propagation Semantics

We are now able to define the denotational higher order demand propagation semantics as an interpretation of the core language, taking the domains of demand propagators as the non-standard semantic domains. The formal definition is listed in Fig. 2.

Most defining rules of the expression semantics are straightforward parallel to the definitions of the standard semantics. Two notes about case-expressions and user type definitions will emphasize the peculiarities:

Fig. 2. Demand Propagation Semantics (cont.)

$\underline{\mathcal{E}} : \forall\tau.\ \mathsf{Expr}^\tau \to \underline{\mathsf{Env}} \to \mathsf{Propagator}^\tau$ where $\underline{\mathsf{Env}} = \forall\tau.\ \mathsf{Var}^\tau \oplus \mathsf{Const}^\tau \to \mathsf{Propagator}^\tau$

$\underline{\mathcal{E}}[\![x]\!]\ \rho\ \Delta = \rho\ x\ \Delta$ þþþþand þþþþ $\underline{\mathcal{E}}[\![C]\!]\ \rho\ \Delta = \rho\ C\ \Delta$

$\underline{\mathcal{E}}[\![e_1\ e_{2]\!]}\ \rho\ \Delta = \underline{\mathcal{E}}[\![e_{1]\!]}\ \rho\ \Delta\ (\underline{\mathcal{E}}[\![e_{2]\!]}\ \rho)$

$\underline{\mathcal{E}}[\![\lambda x.\ e]\!]\ \rho\ \Delta = \underline{\lambda x}.\ \underline{\mathcal{E}}[\![e]\!]\ \rho[\underline{x}/x]\ \Delta$

$\underline{\mathcal{E}}[\![\textbf{case}\ e\ \textbf{of}\ C_1\ v_{11} \ldots v_{1a1} \Rightarrow e_1;\ \ldots;\ C_n\ v_{n1} \ldots v_{nan} \Rightarrow e_n;\ v \Rightarrow e_{def]\!]}\ \rho\ \Delta$

$= \underline{\mathcal{E}}[\![e]\!]\ \rho\ \mathsf{WHNF}\ \&\ (\pi_1 \mid \ldots \mid \pi_n \mid \pi_{def})$

where $\pi_i = \mathit{lift}^\tau(\underline{\mathcal{E}}[\![e]\!]\ \rho\ (C_i\ \mathsf{NO} \ldots \mathsf{NO}))\ \&\ \underline{\mathcal{E}}[\![e_{i]\!]}\ \rho[\underline{v}_{i1}/v_{i1},\ \ldots,\ \underline{v}_{iai}/v_{iai}]\ \Delta$

$\underline{v}_{ij} = \lambda\Delta.\ \underline{\mathcal{E}}[\![e]\!]\ \rho\ (C_i\ \mathsf{NO} \ldots \Delta \ldots \mathsf{NO})$, with Δ at j-th position

$\pi_{def} = \underline{\mathcal{E}}[\![e_{def]\!]}\ \rho[(\lambda\Delta.\ \underline{\mathcal{E}}[\![e]\!]\ \rho\ \Delta \backslash \{C_1, \ldots, C_n\})/v]\ \Delta$

$\underline{\mathcal{E}}[\![\textbf{let}\ m\ \textbf{in}\ e]\!]\ \rho\ \Delta = \underline{\mathcal{E}}[\![e]\!]\ (\underline{\mathcal{M}}[\![m]\!]\ \rho)\ \Delta$

$\underline{\mathcal{U}} : \mathsf{Usertype} \to \underline{\mathsf{Env}} \to \underline{\mathsf{Env}}$

$\underline{\mathcal{U}}[\![\forall\alpha_1 \ldots \alpha_m.\ C_1\ \tau_{11} \ldots \tau_{1a1} + \ldots + C_n\ \tau_{n1} \ldots \tau_{nan]\!]} = [\underline{C}_1/C_1,\ \ldots,\ \underline{C}_n/C_n]$

where $\underline{C}_i\ \Delta = \mathit{lift}(\mathsf{FAIL})$, if $\Delta\ v = \bot$ for all v with $\mathit{tag}(v) = C_i$

$\underline{C}_i\ \Delta = \underline{\lambda v}_1.\ \ldots\ \underline{\lambda v}_{ai}.\ (\underline{v}_1\ \Delta{\downarrow_{Ci,1}}\ \&\ \ldots\ \&\ \underline{v}_{ai}\ \Delta{\downarrow_{Ci,ai}})$, otherwise

$\underline{\mathcal{M}} : \mathsf{Module} \to \underline{\mathsf{Env}} \to \underline{\mathsf{Env}}$

$\underline{\mathcal{M}}[\![T_1 = \upsilon_1;\ \ldots;\ T_m = \upsilon_m;\ \mathrm{fdefs}]\!] = \underline{\mathcal{F}}[\![\mathrm{fdefs}]\!]\ (\rho {+\!+} \underline{\mathcal{U}}[\![\upsilon_{1]\!]} {+\!+} \ldots {+\!+} \underline{\mathcal{U}}[\![\upsilon_{n]\!]})$

where $\underline{\mathcal{F}}[\![v_1 = e_1;\ \ldots;\ v_n = e_{n]\!]}\ \rho = \mu P.\ \rho[\underline{\mathcal{E}}[\![e_{1]\!]}\ P/v_1,\ \ldots,\ \underline{\mathcal{E}}[\![e_{n]\!]}\ P/v_n]$

$\underline{F} = \underline{\mathcal{M}}[\![m]\!]\ \rho\ f$ is the semantics of a variable f in a module m with import env. ρ

The semantics of the prelude definitions are given by

$\underline{\mathrm{NUM}}_n\ \Delta = \mathsf{FAIL}$, if $\Delta\ n = \bot$; $= \mathsf{NO}$, otherwise (cf. also $\underline{C}_i$ in the definition of $\underline{\mathcal{U}}$ above)

$\underline{+}\ \Delta = \underline{\lambda X}.\ \underline{\lambda Y}.\ \underline{X}\ \mathsf{WHNF}\ \&\ \underline{Y}\ \mathsf{WHNF}$ þþþþand þþþþ $\underline{\mathrm{BOT}}^\tau\ \Delta = \mathit{lift}^\tau(\mathsf{FAIL})$

1. The propagator for a case-expression propagates a WHNF demand to the expression to be scrutinised and propagates the context demand to all alternatives of the case-expression to build the union of all propagated demands. The propagation to an alternative constrains the scrutinised value to match the pattern of the alternative and must take care of binding it components to the pattern variables.
2. The user type semantics $\mathcal{U}$ introduces a demand propagator for each constructor, which propagates the accordant projections of the context demand to the factors.

The main justification of the definition of higher order demand propagation semantics is its relation to generalised strictness, which is stated in the following section.

4 The Safety of Higher Order Demand Propagation

We have already motivated the notion of *safety* for demand propagators in Example 1 by roughly saying that a safe demand propagator calculates correct generalised strictness information of a function. To formulate and prove the Safety Theorem (Sect. 4.2) a formal definition of safety is provided now.

4.1 Demand Propagators and Generalised Strictness

In Example 1 we motivated how we deduce generalised strictness information of a function, namely by applying the demand propagator of the function to a context

demand and to a combination of the NO and ID propagators, where NO stands for arbitrary arguments and ID for the tested argument. We could call a demand propagator *safe* for a function, if – applied in this way – it delivers correct generalised strictness information for every context demand. Definition 6 is stronger than this, and Corollary 1 shows that it implies the property stated above.

Definition 6 (Safety of a Demand Propagator).. Let $F : \mathsf{D} \to \mathcal{D}[[\tau_1 \to \ldots \to \tau_n \to \tau_{]]}$ (τ a non-functional type) be a function from some domain D into a domain of a type. And let $\underline{F} \in \mathsf{Propagator}^{\tau 1 \to \ldots \to \tau n \to \tau}$ be a demand propagator of this type. Then $\underline{F}$ is called *safe* for F, if for all m (0ðmðn) holds:

$$\text{if for all i (1ðiðm) } \underline{A_i} \text{ is safe for } A_i : \mathsf{D} \to \mathcal{D}[[\tau_{i]]} \text{ then for all } v \in \mathsf{D}: \quad (2)$$
$$(\underline{F}\ \Delta\ \underline{A_1}\ \ldots\ \underline{A_m})\ v = \bot \text{ implies } \Delta\ ((F\ v)\ (A_1\ v)\ \ldots\ (A_m\ v)) = \bot .$$

This recursive definition of safety is well-founded by the same definition reading for non-functional types, where m may only be zero and hence the definition does not depend on the safety of argument propagators.

$\underline{F}$ is defined to be *safe* for a semantic value $f \in \mathcal{D}[[\tau_1 \to \ldots \to \tau_n \to \tau_{]]}$, if it is safe for any function $F = const\ f$, which yields f, ignoring its argument.

Definition 7 (Safety of an Environment). Let $\rho : \mathsf{Env}$ be an environment of semantic values and let $\underline{\rho} : \underline{\mathsf{Env}}$ be an environment of demand propagators. Then $\underline{\rho}$ is defined to be *safe* for ρ, if for all $x \in \mathsf{Vars} \oplus \mathsf{Constructors}$ holds: $\underline{\rho}\ x$ is safe for $\rho\ x$.

Corollary 1 (Deducing Generalised Strictness). Let $\underline{F} \in \mathsf{Propagator}^{\tau 1 \to \ldots \to \tau n \to \tau}$ be safe for $f \in \mathcal{D}[[\tau_1 \to \ldots \to \tau_n \to \tau_{]]}$. For a context demand Δ, $\Delta_i := \underline{F}\ \Delta\ \underline{NO}\ \ldots\ \underline{NO}\ \underline{ID}\ \underline{NO}\ \ldots\ \underline{NO}$ ($\underline{ID}$ at the i-th position of n arguments) is the demand propagated to f's i-th argument. Then f is Δ_i-strict in its i-th argument in a Δ-strict context, meaning:

$$\Delta_i\ v_i = \bot \text{ implies } \Delta\ (f\ v_1\ \ldots\ v_n) = \bot, \text{ for arbitrary } v_j,\ 1ðjðn,\ j \neq i .$$

Proo.f The proposition follows directly from (2), since $\underline{NO}$ is safe for all functions and $\underline{ID}$ is safe for $V_i = \lambda v.\ v$.

4.2 Safety Theorem for the Higher Order Demand Propagation Semantics

Let m be a core language module, ρ an import environment and $\underline{\rho}$ an environment of demand propagators, which is safe for ρ. Then $\underline{\mathcal{M}}[[m]]\ \underline{\rho}$ is safe for $\mathcal{M}[[m]]\ \rho$.

Proof. The proof of this theorem works by proving a slightly stronger proposition by induction on the structure of the declarations in m and carrying the proposition to the fixpoint environment. It is expatiated in [13].

5 Putting Theory into Practice: The Demand Propagation Analysis

The practical strictness analysis using this higher order demand propagation semantics is not yet finished. But a design for the analysis is explained in [12]. The goal of demand propagation analysis is to deduce *safe* demand propagators for syntactic expressions. The propagated demands resulting from an application os a demand propagator to a context demand shall be *as effective as possible*, with the constraints of being still safe and computable. Termination (and also efficiency) of the analysis is obligatory, because its information will be used in the compilation process. It is well known that strictness is not generally decidable, hence one cannot hope to compute the most effective propagated demand for each application of a demand propagator. In fact it is not certain whether such a most-effective safe demand always exists. See [3] for a discussion of this question for projection analysis.

We now give a brief overview of how demand propagation analysis can be made practical (see [12] for a more detailed explanation):

1. From the Safety Theorem we know that higher order demand propagation assigns safe demand propagators to each function in a core language module. Demand propagators are higher order functions. Fortunately there is a known efficient way to calculate higher order functions, namely executing functional programs. Hence we represent the demand propagators in a simple specially suited higher order functional language, the semantics of which is given by the semantics of demand propagators and of the operations on demands and parameterised demands.
2. An operational semantics for this language can then be introduced, which is sound with respect to the denotational demand propagation semantics. And an abstract machine can be defined, which calculates applications of demand propagators to demands and argument propagators, yielding generalised strictness information.
3. The operational semantics is augmented by loop-detection – in a similar way to that described by Hughes and Ferguson [8] – and by approximation to guarantee its termination. This also permits us to adjust the level where approximation takes place, hence trading accuracy of the analysis against its speed.

6 Conclusions and Further Work

A new approach to strictness analysis has been defined. The analysis is capable of deducing generalised strictness for a realistic functional programming language. This is achieved by mapping function definitions to demand propagators by means of a non-standard interpretation. Demand propagators are higher order functions, which propagate context demands to the arguments of the function. Generalised strictness information can be deduced by applying a demand propagator to a context demand and to special arguments. The Safety Theorem guarantees the correctness of this strictness information. The actual computation of demand propagator applications is achieved by generating functional declarations for the demand propagators in a new functional language. Demand propagation can now take place by running programs on a – yet to be formally defined – abstract machine which supports loop-detection and approximation. A prototype implementation of the analysis in Haskell is in development.

As far as we know, higher order demand propagation analysis is the first backward strictness analysis, which can analyse polymorphic *and* higher order functions. This is possible because the new demand propagation language itself is polymorphically typed and higher order. The infinite semantic domains for higher order demand propagation allow very accurate generalised strictness information to be expressed and propagated even across module boundaries, which is difficult in existing strictness analyses, where the information is compiled into annotations to the function's type.

This paper gives just a first introduction to higher order demand propagation. To make it applicable in a real implementation, further work has to be done:

1. The operational semantics of the language used in demand propagation analysis has to be defined. This task is mainly completed and will be published soon together with a proof for the soundness of the operational semantics with respect to the denotational semantics of demand propagators given in this paper.
2. We plan to integrate a prototype of the analysis into a state-of-the-art compiler for a lazy functional language, proving that the analysis is also usable for realistic software engineering. At this stage it will be necessary to identify and attack complexity and efficiency issues of the analysis which has been unattended so far.
3. The examples in the appendix and in [12] show that higher order demand propagation can be more accurate than other existing strictness analyses. A methodical comparison of the power and the complexity of different strictness analyses would be very interesting but also seems to be very difficult.

7 Related Work

The most widely used framework for strictness analysis for lazy functional languages is an *abstract interpretation* introduced by Mycroft [10] and later enhanced for algebraic data types and higher order functions [18,2] and for polymorphism [1]. Evaluators for a type are defined there as special subsets of the standard semantics domain, which is isomorphic to the notion given here. Strictness analysis by abstract interpretation is a forward analysis using finite abstract domains. The strictness information for functions are represented by annotations on the function's type yielding in general poor propagation of strictness information of higher order functions to other modules.

Abstract reduction [11] is a method, which can handle infinite domains and uses a loop-detecting abstract reduction machine. The analysis again is a forward one and transportation of strictness information across module boundaries is also done by type annotations.

Projection analysis was first formulated in [19] for a first order monomorphic language. It was generalised to a higher order (but monomorphic) language [3] and to a polymorphic (but first order) language [1]. Projection analysis works with projections – namely idempotent functions, which approximate the identity. This concept has been proven to be more powerful than the concept of evaluators. For instance it can express general head strictness, which cannot be expressed with demands or evaluators. However the definition of demand propagators given here seems to be independent from the chosen concept of demands. And whether demand propagation can be also based on projections will be examined in further work. Projection analysis uses finite domains for all data types.

Backward analyses using infinite domains have been proposed by Dybjer (*Inverse Image Analysis* [4]), by Hall and Wise [6] and by Tremblay (*Abstract Demand Propagation* [17]). All these analyses are restricted to first order functions.

Another interesting relation of this work is to the proposal of evaluation strategies for parallel programming in [16]. An evaluation strategy is the operational equivalent to a demand defined here. However the intention of the proposal is to allow the programmer to specify evaluation strategies for calculations in a program, and the focus is not on the safety of these strategies. It would be interesting to examine how demand propagation analysis can be used to transform functional programs automatically to programs decorated with safe evaluation strategies and how this implicit parallelisation affects execution time.

References

1. G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions*. Ph.D. Thesis, University of Glasgow, 1993.
2. G. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, 1991.
3. M.K. Davis. *Projection-Based Program Analysis*. Ph.D. Thesis, University of Glasgow, 1994.
4. P. Dybjer. Inverse Image Analysis. In Th. Ottman, editor, *Automata, Languages and Programming*. LNCS 267, pages 21-30, Springer-Verlag, 1987.
5. E. Fehr. *Semantik von Programmiersprachen*. Springer-Verlag, Heidelberg, 1989.
6. C.V. Hall and D.S. Wise. Compiling Strictness into Streams. In *Proceedings - 14th Annual ACM Symposium on Principles of Programming Languages*, pages 132-142, Munich, 1987.
7. M. Horn. *Improving Parallel Implementations of Lazy Functional Languages Using Evaluation Transformers*. Technical Report B 95-15, Dept. of Comp. Science, FU Berlin, 1995.
8. J. Hughes and A. Ferguson. A Loop-Detecting Interpreter for Lazy, Higher-Order Programs. In J. Launchbury and P.M. Sansom, editors, *Functional Programming, Workshops in Computing*, Springer-Verlag, 1992.
9. S. Kamin. Head Strictness Is Not a Monotonic Abstract Property. In *Information Processing Letters*, North Holland, 1992.
10. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. Thesis, University of Edinburgh, 1981.
11. E. Nöcker. *Strictness Analysis Using Abstract Reduction*. Technical Report, University of Nijmegen, 1993.
12. D. Pape. *Higher Order Demand Propagation*. Technical Report B 98-15, Dept. of Comp. Science, FU Berlin, 1998.
13. D. Pape. *The Safety of Higher Order Demand Propagation*. Technical Report B 98-16, Dept. of Comp. Science, FU Berlin, 1998.
    Available at `http://www.inf.fu-berlin.de/~pape/papers/`
14. J. Peterson, K. Hammond, editors, and many authors. *Report of the Programming Language Haskell – A Non-Strict, Purely Functional Language – Version 1.4*.
    Available at `http://www.haskell.org/onlinereport/`
15. S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
16. P. W. Trinder, K. Hammond, et. al. Algorithm + Strategy = Parallelism. In *Journal of Functional Programming*, 8(1), 1998.

17. G. Tremblay. *Parallel Implementation of Lazy Functional Languages Using Abstract Demand Propagation*. Ph.D. Thesis, McGill University Montréal, 1994.
18. P. Wadler. Strictness Analysis on Non-Flat Domains by Abstract Interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis-Horwood, 1987.
19. P. Wadler and R.J.M. Hughes. Projections for Strictness Analysis. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, LNCS 274, Springer-Verlag, 1987.

Appendix: Example Analyses

The relation $\Downarrow$ on parameterised demands defines an operational (reduction-) semantics for demand propagation, which is not formally defined in this paper, but introduced in [12]. The reduction follows algebraic rules, which can easily be proven for the operators on demands and parameterised demands. Some reductions for the recursive examples given here follow from loop-detection and are indicated as such.

Example 2 (Joint Strictness). Imagine that `cond` from Example 1 is used to define the function

```
> uncond = λb.λx. cond b x x
```

In this context of use the second and the third argument of `cond` are always identical. Thus in the application of the demand propagator for `cond` in the demand propagator for `uncond` the context demand is propagated to `x` in both alternatives of the conditional, hence always, implying that `uncond` is strict in both arguments. More: if the context demand is more effective than WHNF, so is the propagated demand (they are identical). The reduction goes as follows

$\underline{\text{UNCOND}}\ \Delta = \lambda\underline{B}.\lambda\underline{X}.\ \underline{\text{COND}}\ \Delta\ \underline{B}\ \underline{X}\ \underline{X} \Downarrow \lambda\underline{B}.\lambda\underline{X}.\ \underline{B}\ \text{WHNF}\ \&\ (\underline{B}\ \text{TRUE}\ \&\ \underline{X}\ \Delta \mid \underline{B}\ \text{FALSE}\ \&\ \underline{X}\ \Delta)$,

hence: $\underline{\text{UNCOND}}\ \Delta\ \underline{\text{NO}}\ \underline{\text{ID}} \Downarrow \underline{\text{NO}}\ \text{WHNF}\ \&\ (\underline{\text{NO}}\ \text{TRUE}\ \&\ \underline{\text{ID}}\ \Delta \mid \underline{\text{NO}}\ \text{FALSE}\ \&\ \underline{\text{ID}}\ \Delta) \Downarrow \text{NO}\&(\Delta|\Delta) \Downarrow \Delta$.

This example shows that so called *joint strictness* can be analysed. But demand propagation is even able to find strictness, which other analyses cannot detect:

Example 3 (Forcing Summands of a Sum). Imagine `cond` to be used in the function

```
> strange = λz.λx.λy. cond e x y
>     where e = case z of 0⇒(case x of 0⇒False;
x⇒True); z⇒True
```

The demand propagator of `cond` states that `e` is demanded with TRUE and `x` is demanded, or it is demanded with FALSE and `y` is demanded. The demand propagator of `e` now states, that demanding `e` with FALSE, demands `x` with WHNF, because the only alternative, in which `e` evaluates to `False` is, if `z` is zero, and in this case `x` will be demanded. The reduction goes as follows:

$\underline{\text{STRANGE}}\ \Delta = \lambda\underline{Z}.\ \lambda\underline{X}.\ \lambda\underline{Y}.\ \underline{\text{COND}}\ \Delta\ \underline{E}\ \underline{X}\ \underline{Y}$

where $\underline{E}\ \Delta = \underline{Z}\ \text{WHNF}\ \&\ (\underline{X}\ \text{WHNF}\ \&\ (\underline{\text{FALSE}}\ \Delta \mid \underline{\text{TRUE}}\ \Delta) \mid \underline{\text{TRUE}}\ \Delta)$

$\Downarrow \lambda\underline{Z}.\ \lambda\underline{X}.\ \lambda\underline{Y}.\ \underline{E}\ \text{WHNF}\ \&\ (\underline{E}\ \text{TRUE}\ \&\ \underline{X}\ \Delta \mid \underline{E}\ \text{FALSE}\ \&\ \underline{Y}\ \Delta)$

$\Downarrow \underline{\lambda Z}.\ \underline{\lambda X}.\ \underline{\lambda Y}.\ \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{Z}\ \mathsf{WHNF}\ \&\ \underline{X}\ \Delta \mid \underline{Z}\ \mathsf{WHNF}\ \&\ \underline{X}\ \mathsf{WHNF}\ \&\ \underline{Y}\ \Delta)$
$\Downarrow \underline{\lambda Z}.\ \underline{\lambda X}.\ \underline{\lambda Y}.\ \underline{Z}\ \mathsf{WHNF}\ \&\ \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{X}\ \Delta \mid \underline{Y}\ \Delta)$,
since $\underline{E}\ \mathsf{WHNF}\ \Downarrow \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{X}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{FALSE}}\ \mathsf{WHNF} \mid \underline{\mathrm{TRUE}}\ \mathsf{WHNF}) \mid \underline{\mathrm{TRUE}}\ \mathsf{WHNF}) \Downarrow \underline{Z}\ \mathsf{WHNF}$
$\underline{E}\ \mathsf{TRUE}\ \Downarrow \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{X}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{FALSE}}\ \mathsf{TRUE} \mid \underline{\mathrm{TRUE}}\ \mathsf{TRUE}) \mid \underline{\mathrm{TRUE}}\ \mathsf{TRUE}) \Downarrow \underline{Z}\ \mathsf{WHNF}$
$\underline{E}\ \mathsf{FALSE} \Downarrow \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{X}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{FALSE}}\ \mathsf{FALSE} \mid \underline{\mathrm{TRUE}}\ \mathsf{FALSE}) \mid \underline{\mathrm{TRUE}}\ \mathsf{FALSE})$
$\Downarrow \underline{Z}\ \mathsf{WHNF}\ \&\ (\underline{X}\ \mathsf{WHNF}\ \&\ (\mathsf{NO} \mid \mathsf{FAIL}) \mid \mathsf{FAIL}) \Downarrow \underline{Z}\ \mathsf{WHNF}\ \&\ \underline{X}\ \mathsf{WHNF}$.
It follows: $\underline{\mathrm{STRANGE}}\ \Delta\ \underline{\mathrm{ID}}\ \underline{\mathrm{NO}}\ \underline{\mathrm{NO}} \Downarrow \underline{\mathrm{ID}}\ \mathsf{WHNF}\ \&\ \underline{\mathrm{NO}}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{NO}}\ \Delta \mid \underline{\mathrm{NO}}\ \Delta) \Downarrow \mathsf{WHNF}$
$\underline{\mathrm{STRANGE}}\ \Delta\ \underline{\mathrm{NO}}\ \underline{\mathrm{ID}}\ \underline{\mathrm{NO}} \Downarrow \underline{\mathrm{NO}}\ \mathsf{WHNF}\ \&\ \underline{\mathrm{ID}}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{ID}}\ \Delta \mid \underline{\mathrm{NO}}\ \Delta) \Downarrow \mathsf{WHNF}$
$\underline{\mathrm{STRANGE}}\ \Delta\ \underline{\mathrm{NO}}\ \underline{\mathrm{NO}}\ \underline{\mathrm{ID}} \Downarrow \underline{\mathrm{NO}}\ \mathsf{WHNF}\ \&\ \underline{\mathrm{NO}}\ \mathsf{WHNF}\ \&\ (\underline{\mathrm{NO}}\ \Delta \mid \underline{\mathrm{ID}}\ \Delta) \Downarrow \mathsf{NO}$.

Hence demand propagation analysis can infer, that `strange` is strict in its first *and in its second* argument. No other strictness analysis we know would have found the strictness in the second argument.

Note that the demand propagators $\underline{\mathrm{TRUE}}$ and $\underline{\mathrm{FALSE}}$ are not special but are defined implicitly by the semantics $\underline{\mathcal{U}}$ for the user defined data type Bool = False + True. Hence similar examples can be constructed for arbitrary user defined data types.

Example 4 (Construction of Conforming Demands). This example shows, how demands are constructed by the demand propagation analysis, yielding demands, which are not in the abstract domains for existing strictness analyses. Let the function `sum2` be defined by

```
> sum2 = λxs. case xs of  Nil ⇒ 0
>      Cons a t ⇒ case t of   Nil ⇒ 0
>                      Cons b t ⇒ a+b
```

The demand propagator for `sum2` is given by the following equation:
$\underline{\mathrm{SUM2}}\ \Delta = \underline{\lambda XS}.\ \underline{XS}\ \mathsf{WHNF}\ \&$
$(\underline{XS}\ \mathsf{NIL} \mid \underline{XS}\ (\mathsf{CONS}\ \mathsf{NO}\ \mathsf{NO})\ \&\ (\underline{T}\ \mathsf{NIL} \mid \underline{T}\ (\mathsf{CONS}\ \mathsf{NO}\ \mathsf{NO})\ \&\ \underline{A}\ \mathsf{WHNF}\ \&\ \underline{B}\ \mathsf{WHNF}))$
where $\underline{T}\ \Delta = \underline{XS}\ (\mathsf{CONS}\ \mathsf{NO}\ \Delta)$, $\underline{A}\ \Delta = \underline{XS}\ (\mathsf{CONS}\ \Delta\ \mathsf{NO})$ and $\underline{B}\ \Delta = \underline{T}\ (\mathsf{CONS}\ \Delta\ \mathsf{NO})$
$\Downarrow \underline{\lambda XS}.\ (\underline{XS}\ \mathsf{NIL} \mid \underline{XS}\ (\mathsf{CONS}\ \mathsf{NO}\ \mathsf{NIL}) \mid \underline{XS}\ (\mathsf{CONS}\ \mathsf{WHNF}\ (\mathsf{CONS}\ \mathsf{WHNF}\ \mathsf{NO}))$,
hence: $\underline{\mathrm{SUM2}}\ \Delta\ \underline{\mathrm{ID}} \Downarrow \mathsf{NIL} \mid (\mathsf{CONS}\ \mathsf{NO}\ \mathsf{NIL}) \mid (\mathsf{CONS}\ \mathsf{WHNF}\ (\mathsf{CONS}\ \mathsf{WHNF}\ \mathsf{NO}))$.

An effective demand on `sum2` propagates to the demand "evaluate the first and the second list-element to weak head normal form", which is neither an element of the 4-point-list-domain used in [18] or [2], nor of the finite domain of projections for lists used in [3]. Though it could be defined in those frameworks, there is no general approach to identify the evaluators, which are well suited for a given program. Those analyses can only detect simple strictness for `sum2`, which turns out to be a loose of information, if for instance the function `plus` should be analysed:

```
> plus = λa.λb. sum2 (Cons a (Cons b Nil))
```

The demand propagator for `plus` is:
$\underline{\mathrm{PLUS}}\ \Delta = \underline{\lambda A}.\ \underline{\lambda B}.\ \underline{\mathrm{SUM2}}\ \Delta\ \underline{C}$
where $c\ \Delta = \mathsf{CONS}\ (\underline{A}\ \Delta{\downarrow_{\mathrm{Cons},1}})\ (\mathsf{CONS}\ (\underline{B}\ \Delta{\downarrow_{\mathrm{Cons},2}}{\downarrow_{\mathrm{Cons},1}})\ \mathsf{NIL})$
$\Downarrow \underline{\lambda A}.\ \underline{\lambda B}.\ (\underline{C}\ \mathsf{NIL} \mid \underline{C}\ (\mathsf{CONS}\ \mathsf{NO}\ \mathsf{NIL}) \mid \underline{C}\ (\mathsf{CONS}\ \mathsf{WHNF}\ (\mathsf{CONS}\ \mathsf{WHNF}\ \mathsf{NO})))$

$\Downarrow \underline{\lambda A}.\ \underline{\lambda B}.\ \underline{A}$ WHNF & $\underline{B}$ WHNF

since $\underline{C}$ NIL $\Downarrow$ FAIL and $\underline{C}$ (CONS NO NIL) $\Downarrow$ FAIL

and $\underline{C}$ (CONS WHNF (CONS WHNF NO)) $\Downarrow$ $\underline{A}$ WHNF & $\underline{B}$ WHNF .

Hence: $\underline{\text{PLUS}}\ \Delta\ \underline{\text{ID}}\ \underline{\text{NO}} \Downarrow$ WHNF

$\underline{\text{PLUS}}\ \Delta\ \underline{\text{NO}}\ \underline{\text{ID}} \Downarrow$ WHNF .

Demand propagation analysis finds correctly, that `plus` is strict in both arguments. If only the simple strictness information of `sum2` had been used, the strictness of `plus` had not been detected, hence `plus` had been compiled without taking the advantage of optimisation.

Example 5 (Higher Order, Polymorphic Analysis). Define the higher order and polymorphic function `map` by:

```
> map = λf.λxs. case xs of   Nil ⇒ Nil
>       Cons h t ⇒ Cons (f h) (map f t)
```

The demand propagator for `map` is given by the following equation:

$\underline{\text{MAP}}\ \Delta = \underline{\lambda F}.\ \underline{\lambda XS}.\ \underline{XS}$ NIL | $\underline{XS}$ (CONS NO NO) & $\underline{F}\ \Delta\downarrow_{\text{Cons},1}\ \underline{H}$ & $\underline{\text{MAP}}\ \Delta\downarrow_{\text{Cons},2}\ \underline{F}\ \underline{T}$

where $\underline{H}\ \Delta = \underline{XS}$ (CONS Δ NO) and $\underline{T}\ \Delta = \underline{XS}$ (CONS NO Δ) .

In the following reduction we infer the amount of strictness in the second argument in the context of a SPINEELEM demand (SPINEELEM = CONS WHNF SPINEELEM). For illustration, $\underline{F}$ remains a variable in this reduction:

$\underline{\text{MAP}}$ SPINEELEM $\underline{F}$ $\underline{\text{ID}}$ $\Downarrow$ NIL | (CONS NO NO) & $\underline{F}$ WHNF $\underline{H}$ & $\underline{\text{MAP}}$ SPINEELEM $\underline{F}$ $\underline{T}$

$\Downarrow$ NIL | CONS ($\underline{F}$ WHNF $\underline{\text{ID}}$) ($\underline{\text{MAP}}$ SPINEELEM $\underline{F}$ $\underline{\text{ID}}$) .

In the general case, when nothing is known about `f`, we set $\underline{F} = \underline{NO}$ and find:

$\underline{\text{MAP}}$ SPINEELEM $\underline{\text{NO}}$ $\underline{\text{ID}} \Downarrow$ NIL | CONS NO ($\underline{\text{MAP}}$ SPINEELEM $\underline{\text{NO}}$ $\underline{\text{ID}}$) $\Downarrow$ SPINE .

The last reduction is done by loop-detection and identifying the recursive pattern of SPINE = CONS NO SPINE. The same is done in the following reductions.

If strictness of `f` is known ($\underline{F}$ WHNF $\underline{\text{ID}} \Downarrow$ WHNF), we can infer:

$\underline{\text{MAP}}$ SPINEELEM $\underline{F}$ $\underline{\text{ID}} \Downarrow$ NIL | CONS WHNF ($\underline{\text{MAP}}$ SPINEELEM $\underline{F}$ $\underline{\text{ID}}$) $\Downarrow$ SPINEELEM .

And finally, if even more strictness of `f` is known (e.g. for `f = length` with $\underline{\text{LENGTH}}$ WHNF $\underline{\text{ID}} \Downarrow$ SPINE), then the propagated demand is again more informative:

$\underline{\text{MAP}}$ SPINEELEM $\underline{\text{LENGTH}}$ $\underline{\text{ID}} \Downarrow$ NIL | CONS SPINE ($\underline{\text{MAP}}$ SPINEELEM $\underline{\text{LENGTH}}$ $\underline{\text{ID}}$) $\Downarrow$ SPINESPINE .

All this information is captured in the demand propagator $\underline{\text{MAP}}$ for `map`, and since this demand propagator is represented as a function in the language of demand propagation analysis, this information can be used for any special use of `map` in the program, even in another module. The latter were not possible, if strictness is a flat annotation to the type signature of the function, as usually.