Higher Order Demand Propagation

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

Abstract. In this report a new backward strictness analysis for functional languages is presented. It is called higher order demand propagation and is applicable to a realistic non-strict functional language, which has a polymorphic type system and supports higher order functions and user definable algebraic data types. This report defines a semantics for higher order demand propagation and relates it to the standard semantics of the functional language. Each definition in a program is mapped to a demand propagator, which is a higher order function, that propagates context demands to function arguments. The strictness information deduced by the 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 the type information 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. An approach to integrate the higher order demand propagation analysis into an existing compiler for a lazy functional language is sketched at the end of the report.

1 Introduction

This report describes work in progress in the domain of strictness analysis for lazy functional languages. Strictness analysis is one of the major techniques used in optimising compilers for lazy functional programs. If a function is identified to be strict in some arguments, these arguments can 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 arguments of function calls can be reduced independently.

Generalised strictness analysis cannot only deduce that arguments are needed and hence will surely be evaluated. But it can also derive an amount of evaluation, which is safe for a larger data structure. Such information can be used to find larger parallel tasks in a parallel implementation ([Bu91] or [Ho95]). Though it is not always recommended to use generalised strictness information for a sequential implementation – because of the danger to introduce space leaks – it can in general improve simple strictness analysis results even in the sequential setting.

In this paper a new approach to backward general strictness analysis is proposed. The analysis is called demand propagation because evaluation demands are propagated from composed expressions to its components. In contrast to other approaches to backward analysis like projection analysis (Wadler and Hughes [WH87], Davis [Da94]) or abstract demand propagation (Tremblay [Tr94]) this analysis is applicable for a higher order *and* polymorphic language and works with infinite domains in the non-standard semantics.

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

Demands and demand propagators are defined and the demand propagation semantics is presented in section 3. This non-standard semantics assigns a safe demand propagator to each expression. The role of demand propagators in generalised strictness analysis and the notion of safety are defined in section 4.

The demand propagation semantics departs from other abstract interpretations for strictness analysis. Its abstract values are not equivalence classes of standard values but higher order functions, which can calculate the generalised strictness information via backward propagation of demands. A serious implication of this is, that the domains in demand propagation semantics are infinite. This has the advantage, that more accurate strictness information of a function can be expressed and made available to functions that are composed of it, even across module boundaries. This can be seen by looking at the examples in the appendices. On the other hand the analysis has to deal with those infinite domains, and its termination as well as efficient calculation have to be guaranteed. An approach to achieve this is sketched in section 5 and needs to be refined in further work. It includes the possibility to trade accurateness against speed, making the analysis applicable for different stages of program development.

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

2 Core Language Syntax and Semantics

A simple but realistic functional language is defined in this section, which bases on 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 not its main expression value, but a transformation of environments. The module's semantics transforms a given *import* environment into an *export* environment by adding to it the denotations of functions, which are defined in the module.

The reason for defining a modular standard semantics is, to be more related to the non-standard semantics defined in section 3. The modular semantics concept is more adequate to formulate the demand propagation semantics, because here we are not only interested in the semantics of one main expression but in those of all 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 summarised in figure 1. It consists of user type declarations and function declarations. Expressions can be built of variables, function applications, lambda abstractions, **case**- and **let**-expressions. Since the language shall be truly higher order, variables can stand for values of all types, including function types. All expressions are assumed to be typed, but for simplicity all type annotations are omitted in this paper. In a real implementation the principle types can be inferred by Hindley-Milner type inference. Nevertheless we define a type language, which is used to index the semantic domains assigned to the types. 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 definitions of user defined type constructors. This is a usual approach, e. g. followed in the definition of the functional language Haskell ([PH97]). Such a polymorphic type can be applied to a number of types yielding a specialised type, which yet may or may not be polymorphic. Constructors of an algebraic data type are assumed to be unique over all types.

Constants are not incorporated into the expression language. Instead they are assumed to be declarations in a prelude module. Thus the usual constants 0, 1, -1, …, +, *, … appear in the language as variables with their semantics assigned to them in a prelude environment. Integer numbers on the other hand also represent constructors, which tag 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.

The **case**-expression in the core language always contains 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 a default alternative with the expression **bot**, where **bot** is defined in the prelude environment for all types with the semantics $\perp$.

Type language

type :== [∀ { **typevar** } **.**] *monotype*
monotype :==**typevar** | *flat* | *functional* | *typeapplication*
flat :== **INT**
functional :== *monotype* → *monotype*
typeapplication :== **typeconstructor** { *monotype* }

User type declaration language

usertype :== [∀ { **typevar** } **.**] *algebraic*
algebraic :== [*algebraic* **+**] **constructor** { *monotype* }

Expression language

expression :== **var** | **constructor**
| *expression expression*
| λ **var .** *expression*
| **case** *expression* **of** { **constructor** { **var** } ⇒ *expression* } **var** ⇒ *expression*
| **let** *module* **in** *expresssion*

Module language

module :== { **typeconstructor =** *usertype* } { **var =** *expression* }

Figure 1. Core Language Syntax

2.2 Standard Semantics of Core Language Modules

The definition of the standard semantics is given in figure 2 and consists of four parts:

1. **The type semantics** $\mathcal{D}$**:** the semantic domains are indexed by the types of the type language. They are constructed out of the flat domain for INT by function space construction for functional types and sum-of-products construction for user defined algebraic data types. Function domains are generally lifted to contain a least element $\perp$, denoting functional expressions, which have no 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. The domains for user defined types are represented by polymorphic type constructors, which can be applied to the appropriate number of types and yield the domains for recursive data types, which themselves are defined as fixpoints of domain equations in the usual way ([Fe89]). For simplicity and because we are mainly interested in the expression semantics we do not repeat the discussion of recursive domain equations here, but we handle the typeconstructors as if they are predefined in the type environment for the type semantics.

2. **The expression semantics** $\mathcal{E}$**:** Syntactic expressions of 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 (functional) expressions are families of functions for all possible 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 summands tagged with the constructor C and with projection functions $proj_{\mathsf{C},\mathrm{i}}$ to the i-th factor of the summand tagged with C. The injection functions can explicitly be used in the functional program and their semantics are provided by the user type declaration semantics. The projection functions are only used implicitly in the semantics of the **case**-expression. For convenience we also define a function tag, which maps any non-bottom value of a sum to the tag of the summand it belongs to. For $v \neq \perp$ it holds $v \in Image(in_{tag(v)})$.

   The deconstructor **case** could be defined as a family of implicit functions for each user type. But here it is included as a language construct. The reason for this is, that with this notion the **case**-selection on the (infinite) data type INT can be handled in the same manner as scrutinisation on an algebraic data structure.

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 recursively cite variables defined in the module itself or in the import environment, referring to their associated semantics. Each variable, which is free in a definition in the module must be defined somewhere in the module itself or must have a semantics assigned to it by the import environment. The semantics of the set of mutual recursive declarations in a module is the (always existing) minimal fixpoint of an equation describing the transformation of environment transformations.

See figure 2 for the formal definitions of the standard semantics functions.

Domains associated to types

$\mathcal{D}$: Types $\rightarrow$ Typeenvironment $\rightarrow$ Domains

where Typeenvironment := (Typevars $\oplus$ Typeconstructors) $\rightarrow$ Domains

$\mathcal{D}[\![\forall\alpha_1\ldots\alpha_n.\ \tau]\!]\ \sigma = (D_1,\ldots,D_n) \mapsto \mathcal{D}[\![\tau]\!]\ \sigma[D_1/\alpha_1, \ldots, D_n/\alpha_n]$

$\mathcal{D}[\![\alpha]\!]\ \sigma = \sigma\ \alpha$

$\mathcal{D}[\![\mathbf{INT}]\!]\ \sigma = Z_\perp$

$\mathcal{D}[\![\tau_1 \rightarrow \tau_2]\!]\ \sigma = [\mathcal{D}[\![\tau_1]\!]\ \sigma \rightarrow \mathcal{D}[\![\tau_2]\!]\ \sigma]_\perp$

$\mathcal{D}[\![T\ \tau_1\ldots\tau_n]\!]\ \sigma = \sigma\ T\ (\mathcal{D}[\![\tau_1]\!]\ \sigma,\ldots,\mathcal{D}[\![\tau_n]\!]\ \sigma)$

$\mathcal{D}[\![C_1\ \tau_{11}\ldots\tau_{1a_1} + \ldots + C_n\ \tau_{n1}\ldots\tau_{na_n}]\!]\ \sigma = (D_1 \oplus \ldots \oplus D_n)_\perp$

where $D_i := \mathcal{D}[\![\tau_{i1}]\!]\ \sigma \times \ldots \times \mathcal{D}[\![\tau_{ia_i}]\!]\ \sigma$

Expression semantics

$\mathcal{E}$: Expressions $\rightarrow$ Environment $\rightarrow$ Values

where Environment := (Vars $\oplus$ Constructors) $\rightarrow$ Values

$\mathcal{E}[\![x]\!]\ \rho = \rho\ x$

$\mathcal{E}[\![C]\!]\ \rho = \rho\ C$

$\mathcal{E}[\![e_1\ e_2]\!]\ \rho\ = \perp$, if $\mathcal{E}[\![e_1]\!]\ \rho = \perp$

$= \mathcal{E}[\![e_1]\!]\ \rho\ (\mathcal{E}[\![e_2]\!]\ \rho)$, otherwise

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

$\mathcal{E}[\![\mathbf{case}\ e\ \mathbf{of}\ C_1\ v_{11}\ldots v_{1a_1} \Rightarrow e_1; \ldots; C_n\ v_{n1}\ldots v_{na_n} \Rightarrow e_n; v \Rightarrow e_{def}]\!]\ \rho$

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

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

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

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

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

User type definition semantics (implicit constructor functions)

$\mathcal{U}$: Usertypes $\rightarrow$ Environment

$\mathcal{U}[\![\forall\alpha_1\ldots\alpha_m.\ C_1\ \tau_{11}\ldots\tau_{1a_1} + \ldots + C_n\ \tau_{n1}\ldots\tau_{na_n}]\!] = [c_1/C_1, \ldots, c_n/C_n]$

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

Standard module semantics

$\mathcal{M}$: Modules $\rightarrow$ Environment $\rightarrow$ Environment

$\mathcal{M}[\![T_1 = \upsilon_1; \ldots; T_m = \upsilon_m; \text{fdefs}]\!]\ \rho = \mathcal{F}[\![\text{fdefs}]\!]\ (\rho{+}{+}\mathcal{U}[\![\upsilon_1]\!]{+}{+}\ldots{+}{+}\mathcal{U}[\![\upsilon_n]\!])$

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

Semantics f of a variable f in a module m with import environment ρ_i

$f = \mathcal{M}[\![m]\!]\ \rho_i\ f$

Prelude semantics are provided for integer numbers, + and bot

Figure 2. Standard Semantics of the Core Language

3 Higher Order Demand Propagation Semantics

This section defines the non-standard higher order demand propagation semantics as an abstract interpretation. It introduces a type semantics and an expression semantics as well as the semantics for user type declarations and whole core language modules, following the structure of the definition of the standard semantics.

The semantics of an expression in higher order demand propagation is a demand propagator. A demand propagator is a higher order function, which propagates context demands to subexpressions. It is important to see that the demand propagators can be typed and that their types can be deduced from the types of their inducing expressions. That is especially the case for demand propagators for polymorphic functions, which have themselves a polymorphic type. Latter is the main reason why demand propagation analysis works for a polymorphic language.

The definition for demand propagators, which will be introduced in section 3.2, bases on demands defined as characteristic functions. It can probably be generalised to demands defined as projections as in [WH87], yielding a more powerful analysis, which would be able to analyse e.g. general head strictness. The latter is not possible with the notion of demands used in this paper, nor is it with the standard approach for strictness analysis by abstract interpretation as has been proven in [Ka92]. But since the semantics and the analysis would become more complicated in this generalised case, it is not addressed in this report.

3.1 Demands

A domain of demands Demand^{τ} is defined for each type. A demand for a type τ can be interpreted as an evaluation strategy for values of type τ. In the context of generalised strictness analysis it is sufficient to define it by a characteristic function on the value domain, distinguishing the values for which the evaluation succeeds from those for which it fails.

There are other approaches in the literature for defining demands or so called evaluators. Some will be mentioned and compared in remark 3.1.5.

Definition 3.1.1 (demand). A *demand* of type τ is a continuous function from the standard semantic domain $D[\![\tau]\!]$ to the two-point-domain $2=\{1\}_{\perp}$. Demands represent evaluation strategies like evaluators do in [Bu91]. It maps a semantic value to $\perp$, if and only if the related evaluation strategy fails for that value.

Continuity implies monotonicity, hence if the evaluation fails for some value it also fails for all less defined values. That is what we expect for evaluation strategies.

There are three basic demands, which are fully polymorphic and thus can be applied to all values. Those are namely NO (no evaluation), WHNF (evaluation to weak head normal form) and FAIL (non terminating evaluation). Algebraic demands (e.g. for lists) can be constructed out of component demands. Their definitions are given in figure 3. Note that the evaluation strategy for a constructor demand e.g. $\mathsf{CONS\ WHNF\ NO}$ 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 does not terminate or semantically equivalent yields an error. Such constructor demands arise from the analysis of a **case**-alternative.

Definition 3.1.2 (operators on demands). Demands can be combined by the operators | and &, which are defined as pointwise supremum respectively infimum.

In addition to this, algebraic demands can be *projected* to a component or *restricted* to some summands of the sum. The projected demand is the demand, which is induced on the specified factor of the sum-of-products. Projection is used when analysing constructor applications. The restriction operation restricts the demand to be not in a specified set of summands of a sum. The restricted demand yields $\perp$ on the elements of those summands. Restriction is used to describe the propagation in a default alternative of a **case**-expression. The definitions of the demand operators can be studied in figure 3.

The domain of demands

$\mathsf{Demand}^{\tau} = D[\![\tau]\!] \to 2$

Basic, fully polymorphic demands:

$\mathsf{NO}\ v = 1$, for all v

$\mathsf{WHNF}\ v = \perp$, if and only if $v = \perp$

$\mathsf{FAIL}\ v = \perp$, for all v

Algebraic demand for sum components:

$(\mathsf{C}\ \Delta_1 \ldots \Delta_n)\ v = \inf \{\Delta_1\ v_1, \ldots, \Delta_n\ v_n\}$, if $tag(v)=\mathsf{C}$ hence $v = in_{\mathsf{C}}\ (v_1,\ldots,v_n)$; otherwise $\perp$

Compound demands with & and |:

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

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

Projection to demand components by $\downarrow$:

$\Delta{\downarrow}_{\mathsf{C},i}\ v = \perp$, iff for all $v_1 \ldots v_n$: $\Delta\ (in_{\mathsf{C}}\ (v_1,\ldots,v,\ldots,v_n)) = \perp$ (v at i-th position)

Exclusion of demand components by \\:

$\Delta \backslash \mathsf{CS}\ v\ = \perp$, if $tag(v) \in \mathsf{CS}$

$\qquad\qquad = \Delta\ v$, otherwise

Recursive demands can be defined as fixpoints (examples):

$\mathsf{SPINE} = \mu\Delta.\ \mathsf{NIL} \mid \mathsf{CONS\ NO}\ \Delta$

$(\mathsf{EVEN},\mathsf{ODD}) = \mu(\Delta_1,\Delta_2).\ (\mathsf{NIL} \mid \mathsf{CONS\ NO}\ \Delta_2\ ,\ \mathsf{NIL} \mid \mathsf{CONS\ WHNF}\ \Delta_1)$

Figure 3. Demands and Demand Operators

Some algebraic rules for &, |, $\downarrow$ and \\, which are essential for the demand propagation analysis and which can easily be proven, are summarised in figure 4.

FAIL rules before construction:

$C \ldots \mathrm{FAIL} \ldots = \mathrm{FAIL}$

& and | are commutative and associative, and they distribute.

For all demands Δ hold:

$\mathrm{NO} \mathbin{\&} \Delta = \Delta$

$\mathrm{NO} \mid \Delta = \mathrm{NO}$

$\mathrm{FAIL} \mathbin{\&} \Delta = \mathrm{FAIL}$

$\mathrm{FAIL} \mid \Delta = \Delta$

For effective demands Δ hold:

$\mathrm{WHNF} \mathbin{\&} \Delta = \Delta$

$\mathrm{WHNF} \mid \Delta = \mathrm{WHNF}$

For algebraic demands hold:

$C\ \Delta_1 \ldots \Delta_n \mathbin{\&} C\ \Delta'_1 \ldots \Delta'_n = C\ (\Delta_1 \& \Delta'_1) \ldots (\Delta_n \& \Delta'_n)$

$C\ \Delta_1 \ldots \Delta_n \mathbin{\&} C'\ \Delta'_1 \ldots \Delta'_m = \mathrm{FAIL}$, if $C' \neq C$

$C\ \Delta_1 \ldots \Delta_n \mid C\ \Delta'_1 \ldots \Delta'_n = C\ (\Delta_1 \mid \Delta'_1) \ldots (\Delta_n \mid \Delta'_n)$

For the projection operator ↓ holds:

$(C\ \Delta_1 \ldots \Delta_i \ldots \Delta_n)\downarrow_{C,i} = \Delta_i$, if $\Delta_j \neq \mathrm{FAIL}$ for all $j=1\ldots n$

$(C\ \Delta_1 \ldots \Delta_n)\downarrow_{C',i} = \mathrm{FAIL}$, if $C' \neq C$

$\mathrm{NO}\downarrow_{C,i} = \mathrm{NO}$

$\mathrm{FAIL}\downarrow_{C,i} = \mathrm{FAIL}$

For the restriction operator \ holds:

$(C\ \Delta_1 \ldots \Delta_n)\backslash CS = C\ \Delta_1 \ldots \Delta_n$, if $C \in CS$

$(C\ \Delta_1 \ldots \Delta_n)\backslash CS = \mathrm{FAIL}$, if $C \notin CS$

$\mathrm{NO}\backslash CS = \mathrm{NO}$

$\mathrm{FAIL}\backslash CS = \mathrm{FAIL}$

↓ and \ distribute with & and |:

$(\Delta_1 \mathbin{\&} \Delta_2)\downarrow_{C,i} = \Delta_1\downarrow_{C,i} \mathbin{\&} \Delta_2\downarrow_{C,i}$

$(\Delta_1 \mid \Delta_2)\downarrow_{C,i} = \Delta_1\downarrow_{C,i} \mid \Delta_2\downarrow_{C,i}$

$(\Delta_1 \mathbin{\&} \Delta_2)\backslash CS = \Delta_1\backslash CS \mathbin{\&} \Delta_2\backslash CS$

$(\Delta_1 \mid \Delta_2)\backslash CS = \Delta_1\backslash CS \mid \Delta_2\backslash CS$

Figure 4. Algebraic Rules for &, |, ↓ and \

Remark 3.1.3 (no demands on function domains). The general demands NO, WHNF and FAIL are also defined for function types. But demand propagation semantics is not interested in more complex functional demands which can be imagined. One speciality of the demand propagation semantics is, that only non-functional demands are propagated through the demand propagators. Non-functional demands are those defined for types that do not have "$\rightarrow$" as the outermost type constructor. The demand which shall be propagated by a demand propagator for a function, is always interpreted as a demand on the non-functional result type of the function, which can always be deduced from the function's type. The non-functional result type can be a type variable, hence a freely polymorphic type. However some instances may instantiate the type variable with a functional type. This turns out to be no problem because the propagated demand will also be polymorphic then and can be applied to additional arguments if they are present for that specialised instance of the function.

The result of the propagation is a so called parameterised demand (see definition 3.2.3), which states the dependence of the propagated demand from the further propagation by arguments of the function. The parameterised demand can be seen as a function, which can be applied to the demand propagators of statically known arguments, yielding a result demand (the propagated demand).

This limitation to the propagation of only non-functional demands has also been done for projection analysis by Davis [Da94]. It corresponds with 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.

Paradoxly the lack of functional demands seems to be crucial to make the demand propagation semantics higher order.

Definition 3.1.4 (more-effective relation, effective). The complete partial order of the domain of demands, which is induced by standard function domain construction, has FAIL as its bottom element and NO as a universal greatest element. Since it is somehow counter-intuitive to say that the demand NO is the greatest demand, a new partial order is defined on demands:

A demand Δ is called *more effective* than Δ', noted $\Delta \gg \Delta'$,
if and only if $\Delta < \Delta'$ with respect to the natural c.p.o.
Δ is called *effective* at all, if and only if $\Delta \gg \text{NO}$.

With respect to $\gg$, WHNF is the least effective demand. All other demands are comparable with and strictly greater than NO and WHNF. And they are comparable with FAIL the most effective demand.

Remark 3.1.5 (other demand concepts). In the existing abstract interpretations for strictness analysis, e. g. that defined by Burn ([Bu91]), evaluators for a type are defined as subsets of the standard semantics domain for that type. This is isomorphic to the notion given here as a characteristic functions on the standard semantics domain. In contrast to that, in projection analysis a demand is defined as a projection, that is an idempotent function which approximates the identity. This concept has been proven to be more powerful, because it can express for instance general head strictness which cannot be expressed with the former notion of demands. However the definition of a demand propagator here seems to be independent from the choice of concept for demands and can probably also be based upon a notion of demands as projections. Latter will be examined thoroughly in further work.

3.2 Demand Propagators and Parameterised Demands

The higher order demand propagation semantics maps syntactical function definitions to demand propagators. The latter are higher order functions which propagate context demands to the function's arguments. Since context demands are always interpreted as demands on the non-functional result type of an expression (see remark 3.1.3), the propagation of such a demand to arguments of an expression of function type has to reflect this speciality. In general the demand, which is propagated to a function argument will depend on *all* arguments of the function in at least two senses:

1. It depends on the existence of arguments of the function, since a function application will only be evaluated, if it is saturated with the number of arguments provided in the function's definition.
2. If one argument is itself a function – which is a usual case in higher order functional languages – and if this argument function is applied to the argument, to which the demand shall be propagated, the propagation may depend on the propagation of some demand through this argument function. This is for example the case if the strictness of the higher order function `map` in its second argument is analysed for a strict or a non-strict function as the first argument (see example B.2).

To deal with this situation, the result of a demand propagation for a function is a so called parameterised demand, which states the dependence of the propagated demand from argument demand propagators.

In this sense the demand propagator of a function never yields a propagated demand itself, but it only describes the dependence of the demand propagation from the existence and the value of argument demand propagators. Propagation of a demand to a specific argument of a function – with the goal to deduce strictness information for that argument – must be done by applying the demand propagator of the function to a combination of special argument propagators. This connection between demand propagators and generalised strictness information will be explained in section 4.

Before giving the formal definitions for demand propagators (3.2.2) and parameterised demands (3.2.3), a simple example illustrates, how demand propagation works and introduces a simple notation for parameterised demands, the $\underline{\lambda}$-abstraction.

Example 3.2.1 (Motivation of demand propagators for functions). Let `f` be defined in the core language as follows:

```
> f = λa. λb. a+b
```

then the inferred demand propagator $\textsc{f}$ for `f` is:

$\textsc{f}\ \Delta = \underline{\lambda}\textsc{a}.\ \underline{\lambda}\textsc{b}.\ \textsc{a}\ \mathsf{WHNF}\ \&\ \textsc{b}\ \mathsf{WHNF}$.

$\textsc{f}$ is a function which takes a demand Δ as the first argument and yields a parameterised demand, noted by $\underline{\lambda}$-abstractions. A $\underline{\lambda}$-abstraction can be interpreted as follows: If it is applied to an argument demand propagator, then it is an ordinary lambda abstraction stating a functional dependence of the body from the abstracted argument. In the example above this yields another $\underline{\lambda}$-abstraction. If it is *not* applied it can be read as the propagated demand NO, stating that no demand is propagated to any argument, because the function is provided with an insufficient number of arguments. To achieve this dual semantics the $\underline{\lambda}$-abstraction is defined as a pair of a propagated demand and an ordinary abstraction in definition 3.2.3.

It is important to notice, that the equation for F only holds for effective demands Δ. Precisely, it is taken as the defining equation for the demand propagation on effective non-failure demands. We implicitly assume that each propagator propagates the demand NO to NO and the demand FAIL to FAIL regardless of the arguments of the function, because only such propagators are of interest in generalised strictness analysis.

The explanation above shows, that the equation for F can be unfolded to:

$$\begin{aligned}
&\text{F NO} = (\text{NO}, \lambda\text{A}. (\text{NO}, \lambda\text{B}. \text{NO}))\\
&\text{F FAIL} = (\text{FAIL}, \lambda\text{A}. (\text{FAIL}, \lambda\text{B}. \text{FAIL}))\\
&\text{F } \Delta \quad = (\text{NO}, \lambda\text{A}. (\text{NO}, \lambda\text{B}. \text{A WHNF} \ \& \ \text{B WHNF})), \text{ if } \Delta \gg \text{NO}, \Delta \neq \text{FAIL}.
\end{aligned}$$

If F is applied to an effective non-failure demand and further to two argument demand propagators A and B, the body A WHNF & B WHNF of the abstraction states, that a WHNF demand is propagated to the first and to the second argument.

To test `f`'s strictness in any of the arguments, F Δ is applied to a combination of the special demand propagators NO and ID, specifying propagation by arbitrary arguments or by an argument being tested. If an application of `f` to some statically known arguments is analysed – e. g. in another function definition using `f` – the argument propagators are also statically known and may for instance express the propagation to a commonly shared subexpression.

Observe also that the demand propagator F can be typed and is only defined for context demands Δ of type INT. The same holds for the demand propagator of the function `g` below, which has the same non-functional result type but only one argument:

```
> g = f 1
```

`g` is a partial application of `f` to one statically known argument, and its demand propagator can be inferred by partially applying the demand propagator of `f` (pedantically saying: the second component of the parameterised demand resulting from the demand propagator of `f`) to the demand propagator of the constant `1`, which is constantly NO since `1` has no arguments and no free variables. Hence:

$$\text{G } \Delta = \text{F } \Delta \ 1 = (\underline{\lambda}\text{A}. \underline{\lambda}\text{B}. \text{A WHNF} \ \& \ \text{B WHNF}) \ 1 = \underline{\lambda}\text{B}. 1 \text{ WHNF} \ \& \ \text{B WHNF} = \underline{\lambda}\text{B}. \text{B WHNF}.$$

Definition 3.2.2 (demand propagator, context demand). A *demand propagator* for an expression of type $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'$ (τ' non-functional), is a continuous function, which maps a demand of type τ' (the *context demand*) to a *parameterised demand* (the domain PDemand^τ is defined in definition 3.2.3). The domain of demand propagators of type τ as above is the function space

$$\text{Propagator}^\tau = \text{Demand}^{\tau'} \rightarrow \text{PDemand}^\tau.$$

Since the definition of the domains for parameterised demands itself depends on the domains of demand propagators, this definition and the following definition 3.2.3 are mutually recursive.

Definition 3.2.3 (parameterised demand, propagated demand). A parameterised demand is a pair of a demand (the *propagated demand*) and a continuous function that maps an argument demand propagator to a new parameterised demand. First a draft definition for propagated demands is given, to be refined in the next paragraph:

The domain $\mathsf{PDemand}^{\tau}$ of parameterised demands of type $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'$ (τ' non-functional) is a recursive domain, defined by the equations:

$\mathsf{PDemand}^{\tau'} = \mathsf{Demand}$ where Demand is the union of all Demand^{τ}

$\mathsf{PDemand}^{\tau} = \mathsf{Demand} \times [\mathsf{Propagator}^{\tau_1} \rightarrow \mathsf{PDemand}^{\tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'}]$.

The propagated demand in this draft definition has no specific type, because it can belong to each subexpression of the expression the demand propagator is assigned to. But since all argument propagators must propagate to the same subexpression, all propagated demands must have the same type. Hence, the exact definition introduces the domain of demand propagators for a type $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'$ (τ' non-functional) as a type indexed family of domains, hence a polymorphic domain, which is formally defined – together with a revised definition of the demand propagator domain – as follows:

$\mathsf{PDemand}^{\tau'} = \forall\alpha.\ \mathsf{Demand}^{\alpha}$

$\mathsf{PDemand}^{\tau} = \forall\alpha.\ \mathsf{Demand}^{\alpha} \times [\mathsf{Propagator}^{\tau_1}\ \alpha \rightarrow \mathsf{PDemand}^{\tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'}\ \alpha]$

$\mathsf{Propagator}^{\tau} = \forall\alpha.\ [\mathsf{Demand}^{\tau'} \rightarrow \mathsf{PDemand}^{\tau}\ \alpha]$.

The final definition expresses the constraint, that in order to yield a propagated demand of type α all argument propagators of the parameterised demand must propagate to a demand of type α.

Definition 3.2.4 ($\underline{\lambda}$-abstraction for parameterised demands). The $\underline{\lambda}$-abstraction to define a parameterised demand has already been motivated in example 3.2.1 and is formally defined as follows:

Let $\pi=\pi(\mathrm{P})$ be a parameterised demand, which can depend on a demand propagator P. Then $\underline{\lambda}\mathrm{P}.\ \pi$ denotes the parameterised demand $(\mathsf{NO}, \lambda\mathrm{P}.\ \pi)$. Figure 5 lists the definitions of the demand propagators NO, ID and STRICT, which are essential for general strictness analysis, using the $\underline{\lambda}$-abstraction.

Remark 3.2.5 (notation convention for using parameterised demands). The definition of a parameterised demand for a function type as a pair of a propagated demand and an abstraction reflects that there are two things we want to do with it. The first purpose is to apply it to a demand propagator given for an argument of the function. Applying a parameterised demand always means applying 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 the first component of the parameterised demand. This duality of the parameterised demand corresponds to the concept that functional expressions are first class objects in a functional language. They can be seen as objects themselves (that can be demanded with WHNF for instance) and as functions, which map objects to objects.

$\underline{\lambda}$-abstraction for parameterised demands

$\underline{\lambda}\text{P}.\ \pi(\text{P}) = (\mathsf{NO}, \lambda\text{P}.\ \pi(\text{P}))$

Basic demand propagators NO**,** ID **and** STRICT

$\text{NO}^{\tau}\ \Delta = \mathsf{NO}$, if τ is non-functional

$\text{NO}^{\tau\rightarrow\tau'}\ \Delta = \underline{\lambda}\text{X}.\ \text{NO}^{\tau'}\ \Delta$

$\text{ID}^{\tau}\ \Delta = \Delta$, if τ is non-functional

$\text{ID}^{\tau\rightarrow\tau'}\ \Delta = \text{STRICT}^{\tau\rightarrow\tau'}\ \Delta$

$\text{STRICT}^{\tau}\ \Delta = \mathsf{WHNF}$, if τ is non-functional

$\text{STRICT}^{\tau\rightarrow\tau'}\ \Delta = (\mathsf{WHNF}, \lambda\text{X}.\ \text{STRICT}^{\tau'}\ \Delta)$

Figure 5. Example Demand Propagators

It would be a mess of notation if we always wanted to write the projection to the correct component in the given utilisation of the parameterised demand. Fortunately from the context it is always clear whether the propagated demand or the second component of the pair is used. So the notation can be shortened in both cases: Applying the parameterised demand to a demand propagator means applying its second component. And using it as a demand means using its first component. Two examples for the use of the convention are given:

"$\text{P}\ \Delta\ \text{P}_1\ \text{P}_2$" is the short form for "snd (snd ($\text{P}\ \Delta$) P_1) P_2"

"$\text{P}\ (\text{P}_1\ \Delta)\ \text{P}_2$" is the short form for "snd (P (fst ($\text{P}_1\ \Delta$))) P_2"

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

***op*$\in$ {&, |} is defined on parameterised demands π_i recursively by:**

$\pi_1\ op\ \pi_2 := (\Delta_1\ op\ \Delta_2, f)$

where $(\Delta_i, f_i) = \pi_i$

$f\ \text{X} = f_1\ \text{X}\ op\ f_2\ \text{X}$

projection and restriction on parameterised demands are defined by:

$\pi{\downarrow_{\mathsf{C},i}} := (\Delta{\downarrow_{\mathsf{C},i}}, f')$

where $(\Delta, f) = \pi$

$f'\ \text{X} = (f\ \text{X}){\downarrow_{\mathsf{C},i}}$

$\pi\backslash\mathsf{CS} := (\Delta\backslash\mathsf{CS}, f')$

where $(\Delta, f) = \pi$

$f'\ \text{X} = (f\ \text{X})\backslash\mathsf{CS}$

Figure 6. Generalisation of Demand Constructions to Parameterised Demands

Remark 3.2.6 (generalisation of demand constructions). The operations on demands, which were introduced in definition 3.1.2, generalise in a natural way to parameterised demands by applying them to the first component and recursively to the second (see figure 6). The algebraic rules also generalise naturally.

3.3 Definition of the Demand Propagation Semantics

We are now able to define the denotational demand propagation semantics as an abstract interpretation of the core language, taking the domains of demand propagators as the non-standard semantic domains. The formal definition is listed in figure 7.

Most defining rules of the expression semantics are straightforward parallel to the definitions in the standard semantics. Some notes about application, **case**-expression and user type definitions shall emphasize the particularities:

1. Since the domain of parameterised demands is not an explicitly lifted domain the parameterised demand $\underline{E}[\![e_1]\!]\ \rho\ \Delta$ in the application rule can be directly applied to the argument propagator without having a special case for $\perp$.

2. 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 building the union of all propagated demands. That is correct, since if the evaluation of the scrutinised expression does not fail, one of the alternatives of the **case**-expression is demanded with the context demand.

   The **case**-expression fails, if the evaluation of the expression to scrutinise to constructor normal form fails. The propagation to the alternatives can safely constrain the scrutinised value to match the pattern of the alternative and must take care of binding it components to the pattern variables.

3. A user type definition induces implicitly a demand propagator for each constructor, similar as it has induced a constructor function in the standard semantics. The propagator of a constructor propagates the accordant projections of the context demand to the components of the value.

The main property required for the demand propagation semantics is its safety with respect to the standard semantics. The safety of the demand propagation semantics shall reflect the promise, that if using the information deduced from the demand propagators for changing the evaluation order, this will not alter the semantics of the program.

The next section points out the connection between the demand propagators and generalised strictness information and therefore the opportunities for changing the evaluation order. A safety condition is formulated and proven, which claims the connection between demand propagation semantics and standard semantics of the core language.

4 The Safety of Higher Order Demand Propagation

The safety condition for demand propagation semantics defines the connection between non-standard and standard semantics. In order to be useful for program optimisation the demand propagator assigned to an expression must express correct information about its standard semantics. A demand propagator has to be *safe*

Domains associated to types

$\underline{D}[\![\tau]\!] = \mathsf{Propagator}^{\tau}$

Expression semantics

$\underline{E}[\![x]\!]\ \rho\ \Delta = \rho\ x\ \Delta$

$\underline{E}[\![C]\!]\ \rho\ \Delta = \rho\ C\ \Delta$

$\underline{E}[\![e_1\ e_2]\!]\ \rho\ \Delta = \underline{E}[\![e_1]\!]\ \rho\ \Delta\ (\underline{E}[\![e_2]\!]\ \rho)$

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

$\underline{E}[\![\mathbf{case}\ e\ \mathbf{of}\ C_1\ v_{11}\ldots v_{1a_1} \Rightarrow e_1;\ \ldots;\ C_n\ v_{n1}\ldots v_{na_n} \Rightarrow e_n;\ v \Rightarrow e_{def}]\!]\ \rho\ \Delta$

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

$\quad$ where $\pi_i = \underline{E}[\![e]\!]\ \rho\ (\mathsf{C}_i\ \mathsf{NO}\ldots\mathsf{NO})\ \&\ \underline{E}[\![e_i]\!]\ \rho[\textsc{v}_{i1}/v_{i1},\ \ldots,\ \textsc{v}_{ia_i}/v_{ia_i}]\ \Delta$

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

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

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

User type definition semantics

$\underline{U}[\![\forall\alpha_1\ldots\alpha_m.\ C_1\ \tau_{11}\ldots\tau_{1a_1} + \ldots + C_n\ \tau_{n1}\ldots\tau_{na_n}]\!] = [\textsc{c}_1/C_1,\ \ldots,\ \textsc{c}_n/C_n]$

$\quad$ where $\textsc{c}_i\ \Delta = \mathsf{FAIL}$, if $\Delta\ v = \bot$ for all v with $tag(v)=\mathsf{C}_i$

$\qquad \textsc{c}_i\ \Delta = \underline{\lambda}\textsc{v}_1.\ \ldots\ \underline{\lambda}\textsc{v}_{a_i}.\ (\textsc{v}_1\ \Delta{\downarrow}_{\mathsf{C}_i,1}\ \&\ \ldots\ \&\ \textsc{v}_{a_i}\ \Delta{\downarrow}_{\mathsf{C}_i,a_i})$, otherwise

Standard module semantics

$\underline{M}[\![T_1 = \upsilon_1;\ \ldots;\ T_m = \upsilon_m;\ fdefs]\!] = \underline{F}[\![fdefs]\!]\ (\rho{+}{+}\underline{U}[\![\upsilon_1]\!]{+}{+}\ldots{+}{+}\underline{U}[\![\upsilon_n]\!])$

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

Semantics $\textsc{f}$ of a variable f in a module m with import environment ρ_i

$\textsc{f} = \underline{M}[\![m]\!]\ \rho_i\ f$

Prelude demand propagators

$\textsc{num}_n\ \Delta_m = \mathsf{FAIL}$, if $m\neq n$; where $\Delta_m\ k = 1$, if and only if $k=m$

$\textsc{num}_n\ \Delta_n = \mathsf{NO}$

$\textsc{num}_n\ \mathsf{WHNF} = \mathsf{NO}$

$+\ \Delta = \underline{\lambda}\textsc{x}.\ \underline{\lambda}\textsc{y}.\ \textsc{x}\ \mathsf{WHNF}\ \&\ \textsc{y}\ \mathsf{WHNF}$

$\textsc{bot}^{\tau}\ \Delta = \mathsf{FAIL}$, if τ is non-functional

$\textsc{bot}^{\tau\to\tau'}\ \Delta = (\mathsf{FAIL}, \lambda\textsc{x}.\ \textsc{bot}^{\tau'}\ \Delta)$

Figure 7. Demand Propagation Semantics

for that function, that means the information it provides has to be correct in terms of not altering the standard semantics if it is used in an optimisation procedure.

For understanding the safety condition, which is explicated below, it is necessary to understand, which information will be used how in an optimisation process. In this report the focus is on general strictness information and it is well known how it can safely be used in an optimisation of non-strict functional languages: If a function-argument-combination is known to be strict, the evaluation order can be changed without altering the semantics. The argument can in this case be evaluated prior to the function call, as it would happen in a call-by-value semantics. The generalisation step taken by generalised strictness analysis is that it considers evaluators (or demands) yielding more context information for the argument evaluation (e. g. spine strictness instead of simple strictness). The operational semantics can then specify, that the argument can safely be evaluated to the given amount prior to the function call.

Now we explain the connection between the demand propagators and general strictness information.

4.1 Demand Propagators and Generalised Strictness

Assume we have a function with two arguments and want to deduce generalised strictness information for the second argument if the function call is satisfied. We state that the result of the function call is demanded at least with weak head normal form by applying the demand propagator of the function to the context demand WHNF to propagate this demand to the arguments. The result is a parameterised demand with two argument demand propagators. If we want to know what demand is propagated to the second argument if the first argument is arbitrary, we need to apply the parameterised demand to two special argument propagators expressing the further propagation via the arguments, which are not statically known in this instance. The only safe demand propagator which can be assigned to an arbitrary expression is NO. So we provide the parameterised demand with the first argument NO. Since we are interested in propagation to the second argument we provide it with ID as its second argument, stating that the propagated demand is captured at this argument. For deducing simple strictness it would be sufficient to test with the STRICT propagator in the examined argument.

Hence generalised strictness information of a function can be deduced by applying the demand propagator of the function to the context demand and to a combination of NO and ID propagators, where NO stands for arbitrary arguments and ID for the tested argument. It is easy to see that joint strictness can be inferred in the same way by having more than one ID argument.

The former explanation states the property which we expect from the demand propagators to be useful for generalised strictness analysis. We call a demand propagator with this property *safe*. The definition of safety given below is stronger than the former explanation. But the corollaries 4.1.4 and 4.1.5 show, that this definition produces the property stated above.

Definition 4.1.1 (safety of demand propagators). Let $F : \mathsf{D} \to \mathrm{D}[\![\tau_1\to\ldots\to\tau_n\to\tau]\!]$ (τ a non-functional type) be a so called *dependence*. A dependence is a function generating semantic values from some domain D. Let further be F a demand propagator of the corresponding type, $\mathrm{F}\in \mathsf{Propagator}^{\tau_1\to\ldots\to\tau_n\to\tau}$. Then F is called *safe* for *F*, if for all m, $0\le m\le n$ holds:

$$\text{if for all i, } 1\le i\le m,\ \mathrm{A}_i \text{ is safe for } A_i : \mathsf{D} \to \mathrm{D}[\![\tau_i]\!] \text{ then}$$

$$\text{for all } v\in \mathsf{D}\text{: } (\mathrm{F}\ \Delta\ \mathrm{A}_1\ \ldots\ \mathrm{A}_m)\ v = \bot \text{ implies } \Delta\ ((F\ v)\ (A_1\ v)\ \ldots\ (A_m\ v)) = \bot\ .$$

F is defined to be *safe* for a semantic value $f \in D[\![\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau]\!]$, if it is safe for any dependence F = *const f*, which yields f ignoring the dependence-argument. 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.

The safety theorem in section 4.2 states, that the demand propagators assigned by higher order demand propagation semantics to syntactic function definitions are safe for their standard semantics. But to formulate this theorem, first a notion of safe environments must be defined.

Definition 4.1.2 (safety for environments). Let ρ be an environment of semantic values, hence ρ : (Vars⊕Constructors) → Values and let $\underline{\rho}$ be an environment of demand propagators, hence $\underline{\rho}$: (Vars⊕Constructors) → Propagators. Then $\underline{\rho}$ is defined to be *safe* for ρ, if for all x∈ Vars⊕Constructors holds: $\underline{\rho}$ x is safe for ρ x.

Remark 4.1.3 (safe propagators). There are some trivial conclusions from the safety condition:

1. The demand propagator NO is safe for all dependencies.
2. The demand propagator ID is safe for the dependence $ID = \lambda v.\ v : D \rightarrow D$.
3. Each safe demand propagator must propagate the demand NO to NO.
4. Each demand propagator can safely propagate FAIL to FAIL.

These four conclusions have important implications on the demand propagation analysis, which will be described in section 5 of this report. The first provides an always applicable approximation for use in the analysis. This is crucial for guaranteeing its termination. The first and the second conclusion are used when testing for generalised strictness.

The observations 3 and 4 yield two stationary results of every safe demand propagator, so we only need to define the demand propagators for effective non-failure demands explicitly.

Corollary 4.1.4 (generalised strictness with demand propagators). Let $F \in \text{Propagator}^{\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}$ be safe for $f \in D[\![\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau]\!]$ and for a context demand Δ let Δ_i := F Δ_c NO … NO ID NO … NO (ID at the i-th position of n arguments) be the demand propagated to f's i-th argument. Then f is Δ_i-strict in a Δ-strict context, meaning:

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

The proposition follows directly from definition 4.1.1, since NO is safe for all dependencies and ID is safe for $V_i = \lambda v.\ v$.

Corollary 4.1.5 (simple strictness with demand propagators). With the same premises as above holds: If F WHNF NO … NO STRICT NO … NO » WHNF (ID at the i-th position of n arguments), then f is strict in its i-th argument. This proposition follows directly from corollary 4.1.4, since WHNF v = ⊥, if and only if v = ⊥.

4.2 The Safety Theorem

We are now able to formulate the safety theorem for higher order demand propagation, which states that for each function definition the demand propagation semantics of that function is safe for its standard semantics, assuming an initial safe pair of prelude environments.

Theorem 4.2.1 (safety of demand propagation semantics). Let `f` be the name of a core language function in a core language module m. Let ρ be an import environment and $\underline{\rho}$ an environment of demand propagators, which is safe for ρ. Then $\mathrm{F}{:=}\underline{\mathrm{M}}[\![\mathrm{m}]\!]\ \underline{\rho}$ `f` is safe for $f{:=}\mathrm{M}[\![\mathrm{m}]\!]\ \rho$ `f`.

Proof: The proof of this central theorem works by proving a slightly stronger proposition by induction on the structure of `f`'s definition. It will be published in a further report.

5 Demand Propagation Analysis

The goal of demand propagation analysis is to deduce *safe* demand propagators for syntactic expressions. The propagated demands for each application of the demand propagator shall be *as effective as possible*, with the constraints of being yet safe and computable. Termination (and also small complexity) of the analysis is obligatory, because its information shall 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 sure if such most effective safe demand always exist. See [Da94] for a discussion of this question for projection analysis.

In this section an algorithm for demand propagation analysis is sketched, which yields safe and good approximations of demand propagators for expressions given in the core language syntax.

The denotational semantics of a demand has been defined in definition 3.1.1 as a characteristic function on the value domain. In demand propagation analysis a demand is represented as a finite maybe cyclic graph, describing its construction by algebraic construction and application of &, |, $\downarrow$ and \ out of the basic demands NO, WHNF and FAIL. This construction is similar to that of rational strictness patterns in [HW87]. It is evident that not all demands can be expressed in this way and hence the restriction to this notion will be an approximation of its own. But this is even more the case in the frameworks of abstract interpretation or projection analysis where the abstract domains are restricted to be finite.

It is a speciality of higher order demand propagation that propagated demands can be constructed by the demand propagators and need not be predefined for each type like in existing strictness analyses. These constructed demands conform better with the analysed functions, and more of the approximation takes place in the dynamics of the semantics (in the expression semantics) and not in its statics (that is: the choice of the abstract values for the finite domains).

Because of the infinite semantic domains for demands and demand propagators, it is in general not possible to compute the complete function graph of the demand propagator for a function in finite time as it is done in the standard abstract interpretation approach. The propagated demands have to be computed at the points they are needed for. Demand propagation analysis will achieve that by providing an approximating *operational semantics* for demand propagators. Demand propagators can now define the construction of new demands out of a context demands by following the rules of the higher order demand propagation semantics in figure 7. The algebraic rules presented in figures 3 and 4 can then be interpreted as reduction rules on demands and parameterised demands.

The example strictness derivations in appendix A for non-recursive functions and in appendix B for recursive functions motivate the algorithm of demand propagation analysis. A key feature to get accurate results for the analysis of recursive functions is the detection of recursive patterns in the propagated demand. This can probably be done in a realistic implementation of the analysis in a similar way as Hughes and Ferguson describe in [HF92], where they introduce a loop-detecting interpreter for a lazy functional language. Some changes have to be taken into account, since demands are represented by graphs and not by trees.

In the case where no loop can be detected in a given time, approximation has to be applied to guarantee termination. Fortunately approximation with `NO` is always safely possible, though it in general yields less accurate results.

5.1 Design of a Demand Propagation Analysis Framework

The abstract values of higher order demand propagation are demand propagators, hence higher order functions. It is the task of the demand propagation analysis to generate those functions and apply them to special values for getting the desired strictness information.

Fortunately there is a known efficient way to calculate higher order functions, namely executing higher order functional programs! This leads to the following design for developing a demand propagation analysis framework:

1. An abstract interpretation of the functional core language assigns a demand propagator to each expression. This assignment is known to be safe in the meaning that the demand propagators yield correct generalised strictness information which can be used in program optimisation.
2. Demand propagators are higher order functions. They are represented in a simple specially suited higher order functional language, the semantics of which is given by the semantics of the demand propagators and of the operations on demands and parameterised demands.
3. An operational semantics for demand propagation can then be introduced, which is sound with respect to the denotational demand propagation semantics and which makes it possible to define an abstract machine that calculates demand propagator applications to demands and argument propagators, yielding generalised strictness information.
4. The operational semantics is augmented by loop-detection as well as approximation to guarantee its termination.

In the next part of the section we briefly present a design for the functional language, which is used to express the demand propagators.

5.2 Representation of Demands and Propagators

The higher order functional language for noting demands and demand propagators mainly consists of a list of mutual recursive demand definitions and demand propagator definitions.

The sublanguage for defining demands is capable of expressing finite cyclic graphs of demands constructed out of the basic demands by the standard demand constructions. Cycles are expressed by naming and referring nodes, which are entries of a cycle.

The sublanguage for defining demand propagators consists of a term language to construct parameterised demands and to note a propagator by giving the result parameterised demand in dependence of a symbolic demand Δ. The parameterised demand can be constructed symbolically out of demands using operators and $\underline{\lambda}$-abstraction. A propagator can be applied to a demand yielding a parameterised demand. Changes in the environment are reflected by a **let**-expression for parameterised demands. The complete syntax is summarised in figure 8.

module :== { **dvar =** *demand* } { **pvar** Δ **=** *pdemand* }
demand :== **NO** | **WHNF** | **FAIL** | **dvar** | **constructor** { *demand* }
| *demand op demand* | *demand* ↓ **constructor,int**
| *demand* \ { { **constructor** } }
pdemand :== *demand* | Δ | $\underline{\lambda}$ **pvar .** *pdemand* | *pdemand op pdemand*
| *pdemand* ↓ **constructor,int** | *pdemand* \ { { **constructor** } }
| **pvar dvar** { **pvar** } | **let** *module* **in** *pdemand*
op :== **&** | **|**

Figure 8. Syntax of the Demand Propagation Language

5.3 Analysis by Execution of Demand Propagators

The demand propagation semantics defined in figure 7 can now be treated as a program transformation which transforms a core language module to a module definition in the language for demand propagation analysis. Latter can itself be treated as a functional program with the demand propagation semantics as its denotational semantics.

We know, that given a function `f` in the core language, its strictness properties can be tested by applying the associated demand propagator F to a combination of the special propagators ID and NO. We calculate this application by evaluating the expression F WHNF NO … NO ID NO … NO on the basis of the definitions resulting from the program transformation, checking whether the propagated demand is effective or not and so deducing the generalised strictness of *f* by using corollary 4.1.4.

This can be done by defining an *operational semantics* (for instance a reduction semantics) for the language of demand propagation analysis and proving it sound with respect to its denotational semantics. Latter will be done in a subsequent report. The reduction semantics follows the algebraic rules proven for the operations defined on demands and demand propagators, as well as use some implications from how the safe demand propagators are constructed.

Assuming such a sound operational semantics is successfully defined, an abstract machine can be specified, which is capable of loop-detection and approximation to guarantee termination of each program written in the language. Then we have a powerful tool for strictness analysis. Moreover a trade-off between accuracy and time consumption of the analysis can be easily achieved by tuning the level of recursive reduction before applying approximation.

A possible integration of demand propagation analysis in a real compiler for a functional language is sketched in the following part of the report.

5.4 Integration of the Analysis (Raw Design)

In every compiler for a lazy functional language there is a stage where the source program has been transformed into a semantically equivalent program in a restricted language which is mostly a sublanguage of the core language used in this report. At this stage demand propagation analysis can step in, transforming the core module into a demand propagation language module. This module can actually be compiled and linked together with a run time system and the other modules, which it depends on, to build a program, which is able to calculate propagated demands for functions standing in a demanded context.

To infer strictness information for some functions in the core module the demand propagator program can be queried for the propagated demand for the function's propagator applied to the special values yielding the generalised strictness information. This call of the demand propagator leads to recursive calls of other demand propagators, which it depends on, until a recursive pattern is detected or eventually approximation takes place. The strictness information can then be delivered to further steps of the compilation process, where it finally will be used to optimise the programs operation. This flow of the analysis is sketched in figure 9.

6 Conclusions and Further Work

A new approach to strictness analysis is defined in this report. The analysis is capable of deducing generalised strictness for a serious functional programming language. This is achieved by mapping function definitions to demand propagators by means of an abstract interpretation. Demand propagators are higher order functions which act on demands, propagating them 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

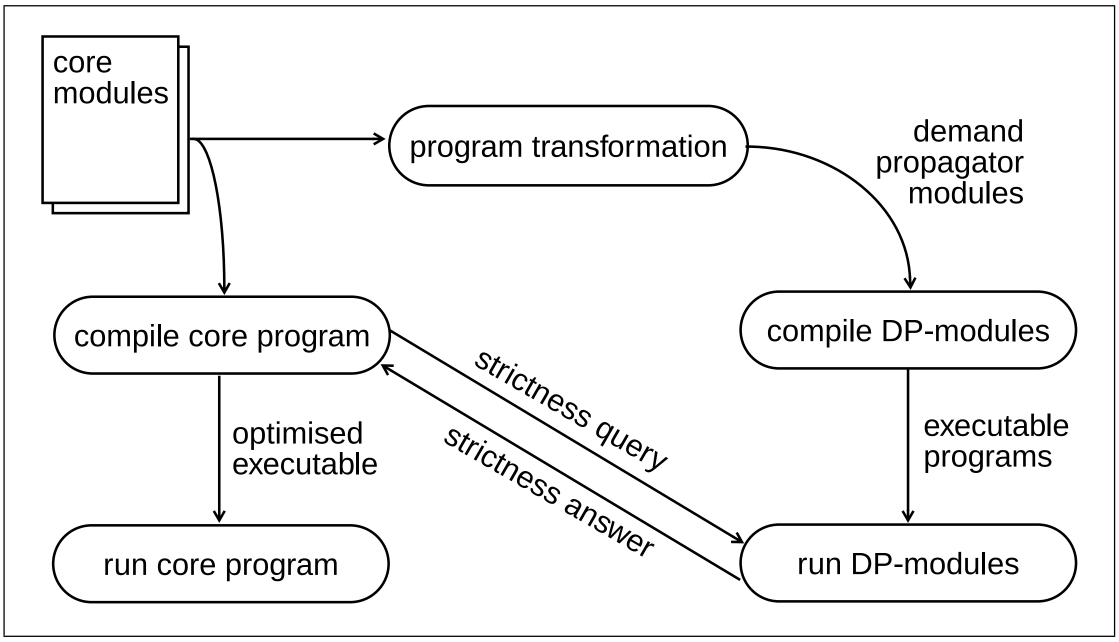

Figure 9. Analysis Flow

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. Latter is difficult in existing implementations of strictness analysis, where the information is compiled into flat annotations to the functions type.

This report is 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 in a subsequent report together with a proof for the soundness of the operational semantics with respect to the denotational semantics of demand propagators given in this report.

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 not only correct but 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. Some examples in the appendices of this report 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. The comparison would be even more interesting, if higher order demand propagators generalise over demands defined as projections (as mentioned in remark 3.1.5).

7 Related Work

The most widely used framework for strictness analysis for lazy functional languages is an abstract interpretation introduced by Mycroft [My81] and later enhanced for algebraic data types and higher order functions (Wadler [Wa87], Burn [Bu91]) and for polymorphism (Baraki [Ba93]). Strictness analysis by abstract interpretation is a forward analysis using finite abstract domains. Strictness information for functions defined in modules are traditionally represented by annotations on the function's type yielding in general poor propagation of strictness information to other modules.

Abstract reduction described by Nöcker [Nö93] is a method that can handle infinite domains and uses a loop-detecting abstract reduction machine. However the analysis is a forward one, and module interfaces only capture selected strictness information yielding poor transportation of strictness information across module boundaries.

Projection analysis was first formulated by Wadler and Hughes [WH87] for a first-order monomorphic language. It was generalised to a higher order (but monomorphic) language by Davis [Da94] and to a polymorphic (but first order) language by Baraki [Ba93]. Projection analysis uses finite domains for all data types.

Backward analyses that use infinite domains have been proposed by Dybjer (Inverse Image Analysis [Dy87]), by Hall and Wise [HW87] and by Tremblay (Abstract Demand Propagation [Tr94]). 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 [TH98]. The definition of an evaluation strategy as a function from some type to the unit type () is semantically equivalent to the definition of demands given here. However the intention of the proposal is to allow the programmer to specify an evaluation strategy for a calculation in a program. The focus is not on the safety of these strategies. The semantics of the enriched program may be different from the semantics of the original program.

It would be interesting to examine how higher order 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

[Ba93] G. Baraki: *Abstract Interpretation of Polymorphic Higher-Order Functions.* Ph.D. Thesis, University of Glasgow 1993

[Bu91] G. Burn: *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman 1991

[Da94] K. Davis: *Projection-Based Program Analysis.* Ph.D. Thesis, University of Glasgow 1994

[Dy87] P. Dybjer: *Inverse Image Analysis.* In Th. Ottman, editor: Automata, Languages and Programming, LNCS 267, Springer 1987

[Fe89] E. Fehr: *Semantik von Programmiersprachen.* Springer, Heidelberg, 1989

[HW87] C. V. Hall and D. S. Wise: *Compiling Strictness into Streams.* Proceedings - 14th Annual ACM Symposium on Principles of Programming Languages, Munich 1987

[Ho95] M. Horn: *Improving Parallel Implementations of Lazy Functional Languages Using Evaluation Transformers.* Technical Report, Freie Universität Berlin 1995

[HF92] J. Hughes and Alex Ferguson: *A Loop-detecting Interpreter for Lazy, Higher-order Programs.* In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Springer 1992

[Ka92] S. Kamin: *Head Strictness is not a monotonic abstract property.* Information Processing Letters, North Holland 1992

[My81] A. Mycroft: *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Thesis, University of Edinburgh 1981

[Nö93] E. Nöcker: *Strictness Analysis using Abstract Reduction.* Technical Report, University of Nijmegen 1993

[PH97] 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/>

[PJ87] S. L. Peyton Jones: *The Implementation of Functional Programming Languages.* Prentice-Hall 1987

[TH98] P. W. Trinder, K. Hammond, et.al. *Algorithm + Strategy = Parallelism.* In Journal of Functional Programming, 8(1), January 1998

[Tr94] G. Tremblay: *Parallel Implementation of Lazy Functional Languages using Abstract Demand Propagation.* Ph.D. Thesis, McGill University Montréal 1994

[Wa87] 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

[WH87] 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 1987

Appendix A (Non-Recursive Examples)

Example A.1: Conditional

Let cond be the conditional function defined by

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

cond has the type $\forall\alpha.\ \mathrm{Bool} \to \alpha \to \alpha \to \alpha$. From the rules in figure 7 we construct the demand propagator COND for cond.

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

An effective demand on the result of the conditional will propagate to one of the arguments x or y and a WHNF demand is propagated to b. The information is even more precise, stating that either b is forced to evaluate to True and the demand is propagated to x, or b is forced to evaluate to False and the demand is propagated to y.

From this information it can safely be deduced that cond is strict in its first argument but not in the others, because:

$$\begin{aligned}
\textsc{cond}\ \Delta\ \textsc{id}\ \textsc{no}\ \textsc{no} &\Downarrow \textsc{id}\ \mathsf{WHNF} \mathbin{\&} (\textsc{id}\ \mathsf{TRUE} \mathbin{\&} \textsc{no}\ \Delta \mid \textsc{id}\ \mathsf{FALSE} \mathbin{\&} \textsc{no}\ \Delta) \\
&\Downarrow \mathsf{WHNF} \mathbin{\&} (\mathsf{NO} \mid \mathsf{NO}) \\
&\Downarrow \mathsf{WHNF} \\
\textsc{cond}\ \Delta\ \textsc{no}\ \textsc{id}\ \textsc{no} &\Downarrow \textsc{no}\ \mathsf{WHNF} \mathbin{\&} (\textsc{no}\ \mathsf{TRUE} \mathbin{\&} \textsc{id}\ \Delta \mid \textsc{no}\ \mathsf{FALSE} \mathbin{\&} \textsc{no}\ \Delta) \\
&\Downarrow \mathsf{NO} \mathbin{\&} (\Delta \mid \mathsf{NO}) \\
&\Downarrow \mathsf{NO} \\
\textsc{cond}\ \Delta\ \textsc{no}\ \textsc{no}\ \textsc{id} &\Downarrow \textsc{no}\ \mathsf{WHNF} \mathbin{\&} (\textsc{no}\ \mathsf{TRUE} \mathbin{\&} \textsc{no}\ \Delta \mid \textsc{no}\ \mathsf{FALSE} \mathbin{\&} \textsc{id}\ \Delta) \\
&\Downarrow \mathsf{NO} \mathbin{\&} (\mathsf{NO} \mid \Delta) \\
&\Downarrow \mathsf{NO}\,.
\end{aligned}$$

But more information (so called joint strictness) is expressed, as can be seen in the next example.

Example A.2: Joint Strictness

Imagine that cond will be used to define another function

```
> uncond = λb.λx. cond b x x
```

In this context of use the second and the third argument of cond are always identical. So 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).

$$\textsc{uncond}\ \Delta = \underline{\lambda}\textsc{b}.\ \underline{\lambda}\textsc{x}.\ \textsc{cond}\ \Delta\ \textsc{b}\ \textsc{x}\ \textsc{x} \Downarrow \underline{\lambda}\textsc{b}.\ \underline{\lambda}\ \textsc{x}.\ \textsc{b}\ \mathsf{WHNF} \mathbin{\&} (\textsc{b}\ \mathsf{TRUE} \mathbin{\&} \textsc{x}\ \Delta \mid \textsc{b}\ \mathsf{FALSE} \mathbin{\&} \textsc{x}\ \Delta)$$

hence:

$$\begin{aligned}
\textsc{uncond}\ \Delta\ \textsc{no}\ \textsc{id} &\Downarrow \textsc{no}\ \mathsf{WHNF} \mathbin{\&} (\textsc{no}\ \mathsf{TRUE} \mathbin{\&} \textsc{id}\ \Delta \mid \textsc{no}\ \mathsf{FALSE} \mathbin{\&} \textsc{id}\ \Delta) \\
&\Downarrow \mathsf{NO} \mathbin{\&} (\Delta \mid \Delta) \\
&\Downarrow \Delta\,.
\end{aligned}$$

This example shows that so called *joint strictness* can be analysed. But the demand propagator contains even more information, which enables demand propagation to infer strictness, which other strictness analyses cannot detect:

Example A.3: Forcing Summands of a Sum

Imagine that cond will be used in the function

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

The demand propagator of cond (as described above) states that e is either forced to evaluate to True and then x is demanded. Or it is forced to evaluate to False and y is demanded. The demand propagator of e now states, that in order to evaluate e, n is demanded with WHNF and in order to evaluate e to False, x is demanded with WHNF, because the only alternative, in which the expression `e` evaluates to `False` is, if `n` is zero, and in this case `x` will be evaluated.

$\mathrm{STRANGE}\ \Delta = \underline{\lambda}\mathrm{N}.\ \underline{\lambda}\mathrm{X}.\ \underline{\lambda}\mathrm{Y}.\ \mathrm{COND}\ \Delta\ \mathrm{E}\ \mathrm{X}\ \mathrm{Y}$

where $\mathrm{E}\ \Delta\ = \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{FALSE}\ \Delta \mid \mathrm{TRUE}\ \Delta)$.

It reduces as follows:

$$\begin{aligned}
\mathrm{STRANGE}\ \Delta &\Downarrow \underline{\lambda}\mathrm{N}.\ \underline{\lambda}\mathrm{X}.\ \underline{\lambda}\mathrm{Y}.\ \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{E}\ \mathsf{TRUE}\ \&\ \mathrm{X}\ \Delta \mid \mathrm{E}\ \mathsf{FALSE}\ \&\ \mathrm{Y}\ \Delta)\\
&\Downarrow \underline{\lambda}\mathrm{N}.\ \underline{\lambda}\mathrm{X}.\ \underline{\lambda}\mathrm{Y}.\ \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{X}\ \Delta \mid \mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{Y}\ \Delta)\\
&\Downarrow \underline{\lambda}\mathrm{N}.\ \underline{\lambda}\mathrm{X}.\ \underline{\lambda}\mathrm{Y}.\ \mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{X}\ \mathsf{WHNF}\ \&\ (\mathrm{X}\ \Delta \mid \mathrm{Y}\ \Delta)
\end{aligned}$$

since

$$\begin{aligned}
\mathrm{E}\ \mathsf{WHNF}\ &\Downarrow \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{FALSE}\ \mathsf{WHNF} \mid \mathrm{TRUE}\ \mathsf{WHNF}) \Downarrow \mathrm{N}\ \mathsf{WHNF}\\
\mathrm{E}\ \mathsf{TRUE}\ &\Downarrow \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{FALSE}\ \mathsf{TRUE} \mid \mathrm{TRUE}\ \mathsf{TRUE})\\
&\Downarrow \mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{TRUE}\ \mathsf{TRUE} = \mathrm{N}\ \mathsf{WHNF}\\
\mathrm{E}\ \mathsf{FALSE}\ &\Downarrow \mathrm{N}\ \mathsf{WHNF}\ \&\ (\mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{FALSE}\ \mathsf{FALSE} \mid \mathrm{TRUE}\ \mathsf{FALSE})\\
&\Downarrow \mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{X}\ \mathsf{WHNF}\ \&\ \mathrm{FALSE}\ \mathsf{FALSE} = \mathrm{N}\ \mathsf{WHNF}\ \&\ \mathrm{X}\ \mathsf{WHNF}\ .
\end{aligned}$$

It follows:

$$\begin{aligned}
&\mathrm{STRANGE}\ \Delta\ \mathrm{ID}\ \mathrm{NO}\ \mathrm{NO} \Downarrow \mathrm{ID}\ \mathsf{WHNF}\ \&\ \mathrm{NO}\ \mathsf{WHNF}\ \&\ (\mathrm{NO}\ \Delta \mid \mathrm{NO}\ \Delta) \Downarrow \mathsf{WHNF}\\
&\mathrm{STRANGE}\ \Delta\ \mathrm{NO}\ \mathrm{ID}\ \mathrm{NO} \Downarrow \mathrm{NO}\ \mathsf{WHNF}\ \&\ \mathrm{ID}\ \mathsf{WHNF}\ \&\ (\mathrm{ID}\ \Delta \mid \mathrm{NO}\ \Delta) \Downarrow \mathsf{WHNF}\\
&\mathrm{STRANGE}\ \Delta\ \mathrm{NO}\ \mathrm{NO}\ \mathrm{ID} \Downarrow \mathrm{NO}\ \mathsf{WHNF}\ \&\ \mathrm{NO}\ \mathsf{WHNF}\ \&\ (\mathrm{NO}\ \Delta \mid \mathrm{ID}\ \Delta) \Downarrow \mathsf{NO}\ .
\end{aligned}$$

TRUE and FALSE are the implicitly defined demand propagators for the algebraic data type `Bool` = `False` | `True`. They can be applied to demands of type Bool, where :

$\mathrm{FALSE}\ \mathsf{FALSE},\ \mathrm{TRUE}\ \mathsf{TRUE},\ \mathrm{FALSE}\ \mathsf{WHNF},\ \mathrm{TRUE}\ \mathsf{WHNF} \Downarrow \mathsf{NO}$

and $\mathrm{FALSE}\ \mathsf{TRUE},\ \mathrm{TRUE}\ \mathsf{FALSE} \Downarrow \mathsf{FAIL}$.

Hence demand propagation analysis can detect 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.

Example A.4: Construction of Conforming Demands

This example will show how demands are constructed by the demand propagation analysis yielding demands, which are not in the abstract domains for existing strictness analyses like abstract interpretation or projection analysis.

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:

$$
\begin{aligned}
\textsc{sum2}\ \Delta = \underline{\lambda}\textsc{xs}.\ & \textsc{xs}\ \mathsf{WHNF} \\
& \&\ (\textsc{xs}\ \mathsf{NIL}\ \mid\ \textsc{xs}\ (\mathsf{CONS\ NO\ NO}) \\
& \qquad \&\ (\textsc{t}\ \mathsf{NIL}\ \mid\ \textsc{t}\ (\mathsf{CONS\ NO\ NO})\ \&\ \textsc{a}\ \mathsf{WHNF}\ \&\ \textsc{b}\ \mathsf{WHNF})) \\
\text{where }\ & \textsc{t}\ \Delta = \textsc{xs}\ (\mathsf{CONS\ NO}\ \Delta) \\
& \textsc{a}\ \Delta = \textsc{xs}\ (\mathsf{CONS}\ \Delta\ \mathsf{NO}) \\
& \textsc{b}\ \Delta = \textsc{t}\ (\mathsf{CONS}\ \Delta\ \mathsf{NO})\ .
\end{aligned}
$$

It reduces as follows:

$$\textsc{sum2}\ \Delta \Downarrow \underline{\lambda}\textsc{xs}.\ (\textsc{xs}\ \mathsf{NIL}\ \mid\ \textsc{xs}\ (\mathsf{CONS\ NO\ NIL})\ \mid\ \textsc{xs}\ (\mathsf{CONS\ WHNF\ (CONS\ WHNF\ NO)}))\ .$$

Hence

$$\textsc{sum2}\ \Delta\ \textsc{id} \Downarrow \mathsf{NIL}\ \mid\ (\mathsf{CONS\ NO\ NIL})\ \mid\ (\mathsf{CONS\ WHNF\ (CONS\ WHNF\ NO)})\ .$$

A WHNF demand on the application of `sum2` propagates to the demand "evaluate the first and the second element of the list to weak head normal form" on its argument. The latter demand is neither an element of the 4-point-list-domain used in abstract interpretations by Wadler [Wa87] or Burn [Bu91], nor of the finite list domain of projections used in projection analysis by Davis [Da94]. Though it could be defined in both of these frameworks, there is no general approach to identify the evaluators, which are well suited for a given program. Those analyses can only infer 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:

$$
\begin{aligned}
& \textsc{plus}\ \Delta = \underline{\lambda}\textsc{a}.\ \underline{\lambda}\textsc{b}.\ \textsc{sum2}\ \Delta\ \textsc{c} \\
& \qquad \text{where } \textsc{c}\ \Delta = \mathsf{CONS}\ (\textsc{a}\ \Delta{\downarrow_{\mathtt{Cons},1}})\ (\mathsf{CONS}\ (\textsc{b}\ \Delta{\downarrow_{\mathtt{Cons},2}}{\downarrow_{\mathtt{Cons},1}})\ \mathsf{NIL})\ .
\end{aligned}
$$

It reduces as follows:

$$
\begin{aligned}
\textsc{plus}\ \Delta & \Downarrow \underline{\lambda}\textsc{a}.\ \underline{\lambda}\textsc{b}.\ (\textsc{c}\ \mathsf{NIL}\ \mid\ \textsc{c}\ (\mathsf{CONS\ NO\ NIL})\ \mid\ \textsc{c}\ (\mathsf{CONS\ WHNF\ (CONS\ WHNF\ NO)})) \\
& \Downarrow \underline{\lambda}\textsc{a}.\ \underline{\lambda}\textsc{b}.\ \textsc{a}\ \mathsf{WHNF}\ \&\ \textsc{b}\ \mathsf{WHNF}
\end{aligned}
$$

since

$$
\begin{aligned}
& \textsc{c}\ \mathsf{NIL} \text{ and } \textsc{c}\ (\mathsf{CONS\ NO\ NIL}) \Downarrow \mathsf{FAIL} \\
& \text{and } \textsc{c}\ (\mathsf{CONS\ WHNF\ (CONS\ WHNF\ NO)}) \Downarrow \textsc{a}\ \mathsf{WHNF}\ \&\ \textsc{b}\ \mathsf{WHNF}\ .
\end{aligned}
$$

Hence:

$$
\begin{aligned}
& \textsc{plus}\ \Delta\ \textsc{id}\ \textsc{no} \Downarrow \mathsf{WHNF} \\
& \textsc{plus}\ \Delta\ \textsc{no}\ \textsc{id} \Downarrow \mathsf{WHNF}\ .
\end{aligned}
$$

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

Appendix B (Recursive Examples)

The analysis of recursive functions makes it necessary to calculate recursive demand propagators and to detect recursive patterns in the propagated demands. This is done in the following examples as stated in their explanations.

Example B.1: Generalised Strictness of `length`

Define the function `length` by:

```
> length = λxs. case xs of
>                   Nil ⇒ 0
>                   Cons h t ⇒ 1 + length t
```

The demand propagator for `length` is defined by the (simplified) equation

$\textsc{length}\ \Delta = \underline{\lambda}\textsc{xs}.\ \textsc{xs}\ \mathsf{WHNF}\ \&\ (\textsc{xs}\ \mathsf{NIL}\ \mid\ \textsc{xs}\ (\mathsf{CONS\ NO\ NO})\ \&\ \textsc{length}\ \mathsf{WHNF}\ \textsc{t})$
    where $\textsc{t}\ \Delta = \textsc{xs}\ (\mathsf{CONS\ NO}\ \Delta)$.

The amount of strictness for the argument of `length` can be correctly inferred as SPINE as the following reduction shows:

$$\begin{aligned}
\textsc{length}\ \mathsf{WHNF}\ \textsc{id}\ &\Downarrow \textsc{id}\ \mathsf{WHNF}\ \&\ (\textsc{id}\ \mathsf{NIL}\ \mid\ \textsc{id}\ (\mathsf{CONS\ NO\ NO})\ \&\ \textsc{length}\ \mathsf{WHNF}\ \textsc{t})\\
&\Downarrow \mathsf{WHNF}\ \&\ (\mathsf{NIL}\ \mid\ (\mathsf{CONS\ NO\ NO})\ \&\ (\mathsf{CONS\ NO}\ (\textsc{length}\ \mathsf{WHNF}\ \textsc{id})))\\
&\Downarrow \mathsf{NIL}\ \mid\ \mathsf{CONS\ NO}\ (\textsc{length}\ \mathsf{WHNF}\ \textsc{id})\\
&\Downarrow \mathsf{SPINE}\ .
\end{aligned}$$

We were using the fact that $\mathsf{CONS\ NO}\ (\textsc{length}\ \mathsf{WHNF}\ \textsc{id})$ is a safe approximation for $\textsc{length}\ \mathsf{WHNF}\ \textsc{t}$ with the given binding $\textsc{t}\ \Delta = \textsc{id}\ (\mathsf{CONS\ NO}\ \Delta)$.

The last reduction follows from matching a recursive pattern: $\textsc{length}\ \mathsf{WHNF}\ \textsc{id}$ is referred to in its own reduction. The observed recursive pattern is that of the SPINE demand.

Example B.2: Higher Order, Polymorphic

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:

$\textsc{map}\ \Delta = \underline{\lambda}\textsc{f}.\ \underline{\lambda}\textsc{xs}.\ \textsc{xs}\ \mathsf{NIL}\ \mid\ \textsc{xs}\ (\mathsf{CONS\ NO\ NO})\ \&\ \textsc{f}\ \Delta{\downarrow_{\mathtt{Cons},1}}\ \textsc{h}\ \&\ \textsc{map}\ \Delta{\downarrow_{\mathtt{Cons},2}}\ \textsc{f}\ \textsc{t}$
    where $\textsc{h}\ \Delta = \textsc{xs}\ (\mathsf{CONS}\ \Delta\ \mathsf{NO})$
        $\textsc{t}\ \Delta = \textsc{xs}\ (\mathsf{CONS\ NO}\ \Delta)$.

Now let us infer the amount of strictness for the second argument in a context of a $\mathsf{SPINEELEM}$ demand ($\mathsf{SPINEELEM} = \mathsf{CONS\ WHNF\ SPINEELEM}$). For illustration $\textsc{f}$ remains a variable in this reduction:

$$\begin{aligned}
&\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{f}\ \textsc{id}\\
&\quad\Downarrow \mathsf{NIL}\ \mid\ (\mathsf{CONS\ NO\ NO})\ \&\ \textsc{f}\ \mathsf{WHNF}\ \textsc{h}\ \&\ \textsc{map}\ \mathsf{SPINEELEM}\ \textsc{f}\ \textsc{t}\\
&\quad\Downarrow \mathsf{NIL}\ \mid\ \mathsf{CONS}\ (\textsc{f}\ \mathsf{WHNF}\ \textsc{id})\ (\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{f}\ \textsc{id})
\end{aligned}$$

`map` can be analysed for generalised strictness in its second argument. We do this first in the general case, when nothing is known about `f` and then in two examples, when generalised strictness information is statically available for `f`.

$\textsc{f} = \textsc{no}$ implies:

$\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{no}\ \textsc{id} \Downarrow \mathsf{NIL}\ \mid\ \mathsf{CONS}\ \mathsf{NO}\ (\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{no}\ \textsc{id}) \Downarrow \mathsf{SPINE}\ .$

Strictness of `f` ($\textsc{f}\ \mathsf{WHNF}\ \textsc{id} \Downarrow \mathsf{WHNF}$) implies:

$\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{f}\ \textsc{id} \Downarrow \mathsf{NIL}\ \mid\ \mathsf{CONS}\ \mathsf{WHNF}\ (\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{f}\ \textsc{id}) \Downarrow \mathsf{SPINEELEM}\ .$

If even more strictness of `f` is known (e. g. `f = length` with $\textsc{length}\ \mathsf{WHNF}\ \textsc{id} = \mathsf{SPINE}$, see example B.1), then the propagated demand is again more informative:

$\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{length}\ \textsc{id}$
$\quad \Downarrow \mathsf{NIL}\ \mid\ \mathsf{CONS}\ \mathsf{SPINE}\ (\textsc{map}\ \mathsf{SPINEELEM}\ \textsc{length}\ \textsc{id}) \Downarrow \mathsf{SPINESPINE}\ .$

The last reductions in each case are done by loop-detection and naming the recursion pattern appropriately.

Example B.3: Analysing Arbitrary User Defined Polymorphic Data Types

This example shows a result for polymorphic algebraic data types. Given the data type of a binary tree, the function `flatten` is analysed, which transforms a binary tree into a list.

```
> data Tree a = Leaf a | Node (Tree a) (Tree a)
>
> flatten = λt. case t of
>                    Leaf v ⇒ Cons v Nil
>                    Node l r ⇒ append (flatten l) (flatten r)
```

Since `flatten` is a polymorphic function, the inferred demand propagator for `flatten` is polymorphic, too. Hence it can be called with any polymorphic demand or with a specialised demand for a specialised use of the function `flatten`.

Demand propagation analysis deduces correctly, that demanding the weak head normal form of the flattened tree, demands the left spine of the tree up to the leftmost leaf. If the head of the flattened tree is demanded with any suited (polymorphic or specialised) demand, higher order demand propagation can infer that the value of the leftmost leaf is demanded with the same demand. The complete analysis however is long and therefore omitted here.

Again the demands inferred here are not in the standard abstract domains of other strictness analyses.