The Safety of Higher Order Demand Propagation

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

Abstract. Higher Order Demand Propagation as proposed in [Pa98] provides a non-standard denotational semantics for a realistic functional language. This semantics can be used to deduce generalised strictness information for higher order polymorphic functions. This report provides the formal proof for the correctness of this strictness information with respect to the non-strict standard semantics.

1 Introduction

A functional core language together with its standard semantics has been introduced in [Pa98]. The language provides higher order functions, a polymorphic type system and user definable recursive data types. A brief summary of the language's syntax and semantics is given in section 2 of this report.

The non-standard semantics of higher order demand propagation has also been declared in [Pa98] and its main definitions are collected in section 3 of this report. The definitions given here are slightly more explicit than those given in [Pa98] – lacking of syntactic sugar – to avoid suggesting implications in the proof just by a sugared notation.

Section 4 contains the formal definition of safety together with the safety theorem. The main implication of the safety theorem is, that generalised strictness information can be obtained from a syntactical function definition by applying its associated demand propagator to a context demand and some special argument demand propagators. This will be the task of the demand propagation analysis, a higher order polymorphic backward strictness analysis, which also has been briefly introduced in [Pa98]. A prototype for this analysis is in implementation and will be released soon.

2 Syntax and Semantics of the Core Language.

A simple but realistic functional language is introduced 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 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 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. 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 syntactic types. Polymorphic types are defined by universal quantification of all free type variables at the outermost level of the type. 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.

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.

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 associated semantics $\perp$.

Type language

type :== [∀ { **typevar** } **.**] *monotype*
monotype :== **typevar** | **INT** | *monotype* → *monotype* | **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** 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 ⊥, 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]). Because we focus on the expression semantics here, we only refer to that it is possible to construct the appropriate domains for recursive data types and handle the typeconstructors as if they are predefined in the type environment for the type semantics.

2. **The expression semantics** 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** U**:** The sum-of-products domains for user defined data types come together with unique continuous injection functions $in_{\mathtt{C}}$ into the sum and with the continuous projection functions $proj_{\mathtt{C},\mathtt{i}}$ to the i-th factor of the summand tagged with C. It is $proj_{\mathtt{C},\mathtt{i}}(v) = \bot$, for all v in other summands than that corresponding to C.

   The injection functions appear as semantics of implicitly defined constructor functions used to construct values in a user defined data type. The projection functions are only used implicitly in the pattern matching semantics of the case-expression. We also define a function *tag*, which maps any non-bottom value of a sum to the constructor name (the tag) of the summand it belongs to and which maps $\bot$ to $\bot$, and we define a function $rest_{\bullet}$. The tag function is only for convenience (though it seems strange to have a function from semantics into syntax). Instead of saying "v is element of the summand corresponding to the constructor C in the domain for the user defined data type T" we can simply write "$tag(v)$=C". The functions $rest_{\bullet}$, parameterised by a set of constructor names CS, are defined by $rest_{\mathtt{CS}}(v)=\bot$, if $v \in$ CS and $=v$, otherwise. They are used in the semantics of the case-expression if the default alternative is matching. They are not really necessary to specify the semantics, but their use simplify the proof of the safety theorem.
4. **The module semantics** M**:** The semantics of a core language module is a transformation of environments. 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 a transformation of environment transformations.

Domains associated to types

D : Types $\to$ Typeenvironment $\to$ Domains

where Typeenvironment = (Typevars $\oplus$ Typeconstructors) $\to$ Domains

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

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

$D[\![\tau_1 \to \tau_2]\!]\ \sigma = [D[\![\tau_1]\!]\ \sigma \to D[\![\tau_2]\!]\ \sigma]_\perp$ and $D[\![T\ \tau_1\ldots\tau_n]\!]\ \sigma = \sigma\ T\ (D[\![\tau_1]\!]\ \sigma,\ldots,D[\![\tau_n]\!]\ \sigma)$

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

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

Expression semantics

E : Expressions $\to$ Environment $\to$ Values

where Environment = (Vars $\oplus$ Constructors) $\to$ Values

$E[\![x]\!]\ \rho = \rho\ x$ and $E[\![C]\!]\ \rho = \rho\ C$

$E[\![e_1\ e_2]\!]\ \rho\ = E[\![e_1]\!]\ \rho\ (E[\![e_2]\!]\ \rho)$, where $\perp\ v = \perp$ for all v

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

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

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

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

$= E[\![e_{def}]\!]\ \rho[rest_{\{C_1,\ldots,C_n\}}(e)/v]$, otherwise[a]

where $e = E[\![e]\!]\ \rho$

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

User type definition semantics (implicit constructor functions)

U : Usertypes $\to$ Environment

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

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

Standard module semantics

M : Modules $\to$ Environment $\to$ Environment

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

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

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

$f = M[\![m]\!]\ \rho_i\ f$

Prelude semantics are provided for integer numbers, + and bot

Figure 2. Standard Semantics of the Core Language

a. In this case is always $rest_{\{C_1,\ldots,C_n\}}(e)=e$, but we write this longer definition for convenient use in the proof of the safety theorem.

3 Demand Propagation Semantics

To understand the demand propagation semantics, it is necessary to provide a notion of demands and demand propagators. The following definitions are taken from [Pa98] and are used throughout the proof of the safety theorem.

Definition 3.1 (Demand)

A *demand* of type τ is a continuous function from the standard semantic domain $D[\![\tau]\!]$ to the two-point-domain $2=\{1\}_\perp$. The continuity of the characteristic function Δ implies the closedness of the set $\Delta^{-1}(\{\perp\})$ with respect to the induced Scott-topology. And the characteristic function Δ_C on a closed set C with $\Delta_C(C) = \perp$ and $\Delta_C(\overline{C}) = 1$ 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 notation because it provides an easy way to define demands on polymorphic domains namely by polymorphic characteristic functions. Operationally demands represent evaluation strategies like evaluators do in [Bu91]. They map a semantic value to $\perp$, if and only if the related evaluation strategy fails for that value.

There are three basic demands, which are fully polymorphic and thus can be applied to all values. Those are 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. 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 3.2 (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 $tag(v) = \mathsf{C}$ and hence $v = in_{\mathsf{C}}(v_1,\ldots,v_n)$; $= \perp$, otherwise

Compound demands with & and |:

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

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

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 $\backslash$:

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

Recursive demands can be defined as fixpoints (examples):

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

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

Figure 3. Demands and Demand Operators

Definition 3.3 (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 $\Delta \gg \mathsf{NO}$.

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.

Definition 3.4 (demand propagator, context demand, parameterised demand, propagated demand)

A *demand propagator* for an expression of type $\tau = \tau_1 \to \ldots \to \tau_n \to \tau'$ (τ' non-functional, as is $\tau = \tau'$, if n=0) is a continuous function, which maps a demand of type τ' (the *context demand*) to a parameterised demand of the same type. The parameterised demand for a non-functional type is just a demand, the so called *propagated demand*. If an expression of a non-functional type has no free variables there will be no target for propagation.[1] Therefore the propagated demand will in general depend on demand propagators assigned to free variables in an expression, which may be bound by a surrounding lambda or defined in the same or another module. The parameterised demand of a function type makes the dependence of the propagated demand from an argument demand propagator explicit, by including a function in its second argument. The first argument is a simple propagated demand, which specifies the propagation if the function is only partially applied.

The domains are defined by the following mutual recursive domain equations:

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

The 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. Note also, that the definition of a parameterised demand of a non-functional type does not depend on the type τ'. The types only apply to the context demand of a propagator and recursively to the context demands of all argument propagators of a parameterised demand. The domains

1. In this case the demand propagator can be used for termination analysis, but this will not be explicated here.

are fully polymorphic in the type for the propagated demand, which reflects, that a demand can be propagated to any subexpression merely by feeding the correct argument demand propagators. How this is done will be described later, when the connection between demand propagators and general strictness is explicated.

The discussion of the types of parameterised demands helps realising, that higher order demand propagation is indeed a polymorphic analysis. Demand propagators of polymorphic functions are also polymorphic. And the proof of safety of a polymorphic demand propagator for a polymorphic function is like a schema of proofs for all instances.

Remark 3.5 (Generalisation of demand constructions)

The operations on demands, which were introduced in definition 3.2, generalise in a natural way to parameterised demands by applying them to the first component and recursively to the second (see figure 4). The both parameterised demands to be combined must have the same type.

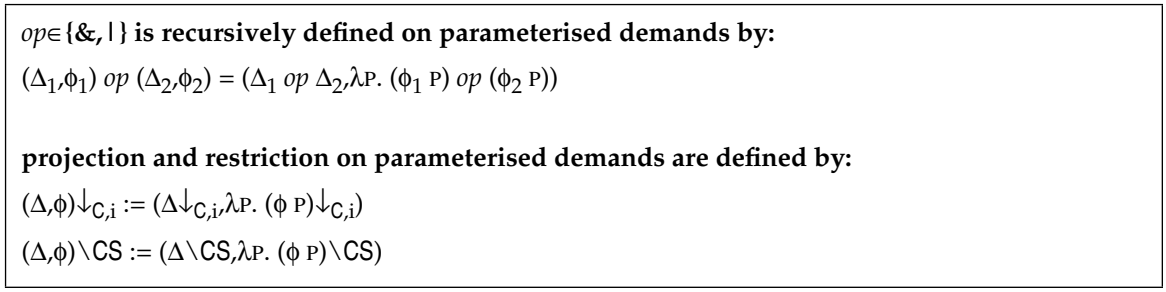

Figure 4. Generalisation of Demand Constructions to Parameterised Demands

Definition 3.6 (Lifting of demand propagators to a functional type)

The case-expression has a special role in our core language, since it triggers evaluation on the expression to be scrutinised. Take the function `f` defined as:

```
> f = λb. case b of True ⇒ λa. a; False ⇒ λa. 1
```

The type of `f` is Bool → Int → Int. The demand propagator `F` of `f` should reflect, that even if `f` is partially applied to only one argument, this argument can safely be demanded with WHNF. In the demand propagation semantics of `f` an application of `b`'s demand propagator to WHNF will be combined with some other propa-

gators for the alternatives by the &-operator. Now these demand propagators *have different types* and thus may not be combined. This dilemma only occurs in the case-expression and fortunately is not a real dilemma. Because the expression to be scrutinised in a case-expression is always of an algebraic (or flat) type its demand propagator maps WHNF to a parameterised demand, which is non-functional, hence does not depend on this type (see definition 3.4). The parameterised demand $\mathtt{b}\,\mathsf{WHNF}$ can easily be lifted to the functional type of the case-expression by the following definition, which describes general lifting of a parameterised demand from an algebraic type to an arbitrary type. Let π be a parameterised demand of algebraic type, then lift^{τ} is defined for each type τ, by:

$\mathit{lift}^{\tau'}(\pi) = \pi$, if τ' is non-functional
$\mathit{lift}^{\tau\to\tau'}(\pi) = (\pi,\lambda x.\ \mathit{lift}^{\tau'}(\pi))$

This definition expresses the intuition, that propagation by a non-functional parameterised demand does not depend on further argument propagators any more, as it is obvious in the above example: The propagation induced by scrutinisation on b does not depend on the propagator provided for a.

The functions *lift* will be used in the demand propagation semantics of the case-expression.

Remark 3.7 (Two ways to apply parameterised demands)
There are two views we want to have on a parameterised demand and therefore two ways to apply it. The first looks at the parameterised demand as a demand. Thus it can be applied to semantic value yielding 1 or $\bot$. The second view gives it its name – parameterised demand – and looks at it as a function, which maps an argument demand propagator to a new parameterised demand, describing the dependence of the propagation by an argument propagator. Thus a parameterised demand can also be applied to a demand propagator.

Though it is always determined by the context, whether a parameterised demand is applied to a semantic value or to an argument propagator, we will distinguish both applications in this paper by different notations to be more explicit in the proof of the safety theorem. We will note the application of a parameterised demand to a demand propagator (second view) by an infix @ or by normal appli-

cation notation, hence by juxtaposition of function and argument. The application of a parameterised demand to a semantic value will be noted by an infix ♦. To be just a little more flexible we will interpret the ♦ as a postfix operator, projecting the parameterised demand to its propagated demand component. Hence the infix ♦ can be alternatively interpreted as a postfix ♦ followed by a normal application of a demand to a semantic value. The formal definitions of @ and ♦ are given in figure 5.

Definition 3.8 ($\underline{\lambda}$-abstraction for parameterised demands)
Let $\pi=\pi(P)$ be a parameterised demand, which can depend on a demand propagator P. Then $\underline{\lambda}P.\ \pi$ denotes the parameterised demand $(NO,\lambda P.\ \pi)$. Figure 5 also lists the definitions of the demand propagators NO, ID and STRICT, which are essential for general strictness analysis.

The duality of parameterised demands

$@ : \mathsf{ParamDemand}^{\tau\to\tau'}\ \alpha \to \mathsf{DemandPropagator}^{\tau}\ \alpha \to \mathsf{ParamDemand}^{\tau'}\ \alpha$

$(\Delta,\phi)\ P = (\Delta,\phi)@P = \phi\ P$

$♦ : \mathsf{ParamDemand}^{\tau}\ \alpha \to \mathsf{Demand}^{\alpha}$

$\Delta♦ = \Delta$

$(\Delta,\phi)♦ = \Delta$

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

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

Basic demand propagators NO, ID and STRICT

$NO^{\tau}\ \Delta = NO$, if τ is non-functional and $NO^{\tau\to\tau'}\ \Delta = \underline{\lambda}P.\ NO^{\tau'}\ \Delta$

$ID^{\tau}\ \Delta = \Delta$, if τ is non-functional and $ID^{\tau\to\tau'}\ \Delta = STRICT^{\tau\to\tau'}\ \Delta$

$STRICT^{\tau}\ \Delta = WHNF$, if τ is non-functional and $STRICT^{\tau\to\tau'}\ \Delta = (WHNF,\lambda P.\ STRICT^{\tau'}\ \Delta)$

Figure 5. Parameterised Demands

Definition 3.9 (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 6. Most defining rules of the expression semantics are straightforward parallel to the definitions in the standard semantics. A user type definition introduces implicitly a demand propagator for each constructor function, similar as it implicitly defined a semantics for the constructor function in the standard semantics. The propagator of a constructor function propagates the accordant projections of the context demand to the components of the value.

Domains associated to types

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

Expression semantics

$\underline{E} : \mathsf{Expressions} \to \mathsf{DPEnvironment} \to \mathsf{Propagators}$

where $\mathsf{DPEnvironment} := (\mathsf{Vars} \oplus \mathsf{Constructors}) \to \mathsf{Propagators}$

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

$\underline{E}[\![e_1\ e_2]\!]\ \rho\ \Delta = \underline{E}[\![e_1]\!]\ \rho\ \Delta\ (\underline{E}[\![e_2]\!]\ \rho)$ and $\underline{E}[\![\lambda x.\ e]\!]\ \rho\ \Delta = \underline{\lambda} x.\ \underline{E}[\![e]\!]\ \rho[x/x]\ \Delta$

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

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

where $\pi_i = \mathit{lift}^{\tau}(\underline{E}[\![e]\!]\ \rho\ (C_i\ \mathsf{NO}\ldots\mathsf{NO}))\ \&\ \underline{E}[\![e_i]\!]\ \rho[v_{i1}/v_{i1}, \ldots, v_{ia_i}/v_{ia_i}]\ \Delta$

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

$\pi_{def} = \mathit{lift}^{\tau}(\underline{E}[\![e]\!]\ \rho\ \mathsf{WHNF}\backslash CS)\ \&\ \underline{E}[\![e_{def}]\!]\ \rho[(\lambda\Delta.\ \underline{E}[\![e]\!]\ \rho\ \Delta\backslash CS)/v]\ \Delta$

$CS = \{C_1, \ldots, C_n\}$

$\tau = \mathit{type}(\mathbf{case}\ e\ \mathbf{of}\ C_1\ v_{11}\ldots v_{1a_1} \Rightarrow e_1; \ldots; C_n\ v_{n1}\ldots v_{na_n} \Rightarrow e_n; v \Rightarrow e_{def})$

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

User type definition semantics

$\underline{U} : \mathsf{Usertypes} \to \mathsf{DPEnvironment}$

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

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

$c_i\ \Delta = \underline{\lambda} v_1.\ \ldots\ \underline{\lambda} v_{a_i}.\ (v_1\ \Delta{\downarrow_{C_i,1}}\ \&\ \ldots\ \&\ v_{a_i}\ \Delta{\downarrow_{C_i,a_i}})$, otherwise

Standard module semantics

$\underline{M} : \mathsf{Modules} \to \mathsf{DPEnvironment} \to \mathsf{DPEnvironment}$

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

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

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

Figure 6. Demand Propagation Semantics

Prelude demand propagators

$\text{NUM}_n\ \Delta$ = FAIL, if Δ n = $\perp$; = NO, otherwise

$\dot{+}\ \Delta = \underline{\lambda}X.\ \underline{\lambda}Y.$ X WHNF & Y WHNF

$\text{BOT}^\tau\ \Delta$ = FAIL

Figure 6. Demand Propagation Semantics (cont.)

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 A Proof for the Safety of Higher Order Demand Propagation

We first define the notion of safety and then show, that it implies the correctness of generalised strictness information deduced from a safe demand propagator. First we proof some lemmata and finally the safety theorem, which states, that each demand propagation semantics of a syntactic function is safe for its standard semantics, assuming a safe pair of prelude environments. The minimal initial prelude consisting of the integer numbers n, and the functions + and bot builds such a safe pair together with the demand propagators NUM_n, $\dot{+}$ and BOT, as proved in lemma 4.3. The safety of the demand propagators, which are implicitly defined for the constructor functions, is also proven in lemma 4.3.

Definition 4.1 (Safety of demand propagators)

We call a function from some domain D to another semantic domain a *dependence* expressing the dependence of a value from exactly one argument.

Let $F : \mathsf{D} \to D[\![\tau_1 \to \ldots \to \tau_n \to \tau]\!]$ (τ a non-functional type) be a *dependence*. And let further be $\mathrm{F} \in \mathsf{Propagator}^{\tau_1 \to \ldots \to \tau_n \to \tau}$ a demand propagator of the corresponding type. Then F is called *safe* for F, if for all m with $0 \le m \le n$ holds:

$$\begin{array}{c} \textit{If } \forall \mathrm{i},\ 1 \le \mathrm{i} \le \mathrm{m}\ .\ \mathrm{A_i} \textit{ is safe for } A_\mathrm{i} : \mathsf{D} \to D[\![\tau_\mathrm{i}]\!] \\ \textit{then } \forall \mathrm{v} \in \mathsf{D}\ .\ (\mathrm{F}\ \Delta\ \mathrm{A}_1\ \ldots\ \mathrm{A}_\mathrm{m})\ \blacklozenge\ v = \bot \Rightarrow \Delta\ ((F\ v)\ (A_1\ v)\ \ldots\ (A_\mathrm{m}\ v)) = \bot \end{array} \quad . \tag{1}$$

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.

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

Definition 4.2 (Safety for environment dependences)
If $\mathrm{P} : \mathsf{D} \to (\mathsf{Vars} \oplus \mathsf{Constructors}) \to \mathsf{Values}$ is a dependence of an environment of semantic values and $\underline{\rho} : (\mathsf{Vars} \oplus \mathsf{Constructors}) \to \mathsf{Propagators}$ an environment of demand propagators. Then $\underline{\rho}$ is defined to be *safe* for P, if

$$\forall \mathrm{x} \in \mathsf{Vars} \oplus \mathsf{Constructors}\ .\ (\underline{\rho}\ \mathrm{x})\ \textit{is safe for}\ (\lambda \mathrm{t}\ .\ \mathrm{P}\ \mathrm{t}\ \mathrm{x})\ . \tag{2}$$

Again $\underline{\rho}$ is defined to be *safe* for an environment of semantic values ρ, if $\underline{\rho}$ is safe for any dependence $\mathrm{P} = \mathrm{const}\ \rho$.

Lemma 4.3 (Some safe demand propagators)
The following propagator safety properties hold:

1. NO is safe for all dependences
2. ID is safe for $id = \lambda v.\ v$
3. for all $n \in Z$: $\mathrm{NUM_n}$ is safe for n
4. $\dot{+}$ is safe for +
5. BOT is safe for $\bot$
6. $\underline{\lambda}\mathrm{v}_1.\ \ldots\ \underline{\lambda}\mathrm{v}_\mathrm{a}.\ (\mathrm{v}_1\ \Delta{\downarrow_{\mathsf{C},1}}\ \&\ \ldots\ \&\ \mathrm{v}_\mathrm{a}\ \Delta{\downarrow_{\mathsf{C},\mathrm{a}}})$ is safe for a constructor function $\lambda v_1.\ \ldots\ \lambda v_\mathrm{a}.\ in_\mathsf{C}(v_1,\ldots,v_\mathrm{a})$

Proof. Let in the following proofs be $\mathrm{v_i}$ safe for V_i for all i and Δ an effective context demand. For the propagators P in each proof define Δ_m to be $(\mathrm{P}\ \Delta\ \mathrm{v}_1\ \ldots\ \mathrm{v}_\mathrm{m})\blacklozenge$.

1. From the definition of NO it follows that (NO Δ v_1 … v_m)♦ = NO regardless of the v_i, hence Δ_m t ≠ ⊥, thus (1) is always true.
2. Assume Δ_m t = ⊥ with Δ_m = (ID Δ v_1 … v_m)♦. Since the definition of ID implies $\Delta_m = \Delta$ » WHNF or Δ_m = WHNF, it follows ⊥ = t = *id* t, hence also Δ (*id* t) = ⊥.
3. Assume (NUM$_n$ Δ)♦t = ⊥. From the definition of NUM$_n$ follows Δ n = ⊥.
4. Since $\Delta_0 = \Delta_1$ = NO, nothing is to show for m=0 or m=1.

   If ⊥ = (+ Δ v_1 v_2)♦t = (v_1 WHNF & v_2 WHNF)♦t, it follows (v_1 WHNF)♦t = ⊥ or (v_2 WHNF)♦t = ⊥. Without loss of generality assume (v_1 WHNF)♦t = ⊥. Since v_1 is safe for V_1, it follows WHNF (V_1 t) = ⊥, and thus V_1 t = ⊥. We then have Δ (+ (V_1 t) (V_2 t)) = Δ (+ ⊥ (V_2 t)) = Δ ⊥ = ⊥, since Δ is effective.
5. It is Δ (⊥ (V_1 t) … (V_m t)) = Δ ⊥ = ⊥, for all m and effective Δ, hence each demand propagator especially BOT is safe for ⊥.
6. If Δ v = ⊥ for all v with *tag*(v)=C, then Δ (*in*$_C$(V_1 t,…,V_a t)) = ⊥ follows immediately for arbitrary t. In the other case assume (v_1 $\Delta\downarrow_{C,1}$ & … & v_m $\Delta\downarrow_{C,a}$)♦t = ⊥.

   It is nothing to show for m<a, since Δ_m = NO then. For m=a, (v_i $\Delta\downarrow_{C,i}$)♦t = ⊥ must hold for at most one i. Without loss of generality assume (v_1 $\Delta\downarrow_{C,1}$)♦t = ⊥. Since v_1 is safe for V_1, it is $\Delta\downarrow_{C,1}$ (V_1 t) = ⊥, hence with the definition of ↓ finally follows Δ (*in*$_C$(V_1 t,…,V_a t)) = ⊥.

q.e.d.

Corollary 4.4 (Generalised strictness with demand propagators)
If F ∈ Propagator$^{\tau_1\to\ldots\to\tau_n\to\tau}$ is safe for $f \in D[\![\tau_1\to\ldots\to\tau_n\to\tau]\!]$, Δ a context demand, and Δ_i = (F Δ NO … NO ID NO … NO)♦ (ID at the i-th position of n arguments). Then f is Δ_i-strict in its i-th argument in a Δ-strict context, meaning:

$$\forall v_1 \ldots v_n \,.\, \Delta_i\ v_i = \bot \Rightarrow \Delta\ (f\ v_1\ \ldots\ v_n) = \bot\ . \tag{3}$$

Proof. Define F = const f, V_j = const v_j for j≠i and V_i = λt. t. From lemma 4.3 we know, that NO is safe for each V_j with j≠i and ID is safe for V_i, hence from (1) we get:
$\Delta_i\ v_i = \bot \Rightarrow \bot = \Delta\ ((F\ v_i)\ (V_1\ v_i)\ \ \ldots\ \ (V_n\ v_i)) = \Delta\ (f\ v_1\ \ldots\ v_n)$.
q.e.d.

Lemma 4.5 (Safety in case alternatives)

If $T = \forall\alpha_1 \ldots \alpha_m.\ C_1\ \tau_{11} \ldots \tau_{1a_1} + \ldots + C_n\ \tau_{n1} \ldots \tau_{na_n}$ is a user defined data type and v is a demand propagator, which is safe for $V : \mathsf{D} \to D[\![T\ \tau_1\ \ldots\ \tau_n]\!]$, then

1. $\lambda\Delta.\ v\ (C\ \mathsf{NO} \ldots \Delta^{(j)} \ldots \mathsf{NO})$ is safe for $(proj_{C,j} \bullet V)$
2. $\lambda\Delta.\ v\ (\Delta \backslash CS)$ is safe for $(rest_{CS} \bullet V)$

Proof.

1. Assume $(v\ (C\ \mathsf{NO} \ldots \Delta^{(j)} \ldots \mathsf{NO})) \blacklozenge t = \bot$. Since v is safe for V, it follows from (1): $(C\ \mathsf{NO} \ldots \Delta^{(j)} \ldots \mathsf{NO})\ (V\ t) = \bot$, hence from the definition of $C\ \mathsf{NO} \ldots \Delta^{(j)} \ldots \mathsf{NO}$, that $tag(V\ t) \neq C$ or $\Delta\ (proj_{C,j}(V\ t)) = \bot$. Both cases imply $\Delta\ (proj_{C,j}(V\ t)) = \bot$.
2. Assume $(v\ (\Delta \backslash CS)) \blacklozenge t = \bot$. Since v is safe for V, it follows from (1): $(\Delta \backslash CS)\ (V\ t) = \bot$, hence from the definition of $\backslash$, that $tag(V\ t) \in CS$ or $\Delta\ (V\ t) = \bot$. Both cases imply $\Delta\ (rest_{CS}\ (V\ t)) = \bot$.

q.e.d.

Lemma 4.6 (Enhancing safe environments)

If $\underline{\rho}$ is safe for P, then the following propositions hold:

1. If F is safe for F, then $\underline{\rho}[F/\mathtt{f}]$ is safe for $\lambda t.\ (P\ t)[F\ t/\mathtt{f}]$
2. If F_i is safe for F_i for all i, $1 \leq i \leq n$, then $\underline{\rho}[F_1/\mathtt{f}_1, \ldots, F_n/\mathtt{f}_n]$ is safe for $\lambda t.\ (P\ t)[F_1\ t/\mathtt{f}_1, \ldots, F_n\ t/\mathtt{f}_n]$

Proof. Let $\underline{\rho}$ be safe for P.

1. For all variables $v \neq \mathtt{f}$ is $\rho[F/\mathtt{f}]\ v = \rho\ v$ safe for $\lambda t.\ P\ t\ v = \lambda t.\ (\lambda t.\ (P\ t)[F\ t/\mathtt{f}])\ t\ v$. And for $\mathtt{f}$ is $\rho[F/\mathtt{f}]\ \mathtt{f} = F$ safe for $F = \lambda t.\ F\ t = \lambda t.\ (\lambda t.\ (P\ t)[F\ t/\mathtt{f}])\ t\ \mathtt{f}$.
2. Follows directly from 1 by induction.

q.e.d.

Lemma 4.7 (Continuity of the safety relation)

If environment dependences P_i are given, such that for each t, $(P_i\ t)$ is a chain. And if $(\underline{\rho}_i)$ is a chain of propagator environments, such that for all i holds: $\underline{\rho}_i$ is safe for P_i, then $\bigcup_i \underline{\rho}_i$ is safe for $\lambda t\ .\ \bigcup_i (P_i t)$.

Proof. Assume $\bigcup_i \underline{\rho}_i$ being not safe for $\lambda t . \bigcup_i (P_i t)$. Then it exists an $f \in \mathsf{Vars}$ with $(\bigcup_i \underline{\rho}_i)\ f$ not safe for $\lambda t . (\bigcup_i (P_i\ t))\ f$. Hence there exist $m \in \mathbb{N}$, v_1, …, v_m, V_1, …, V_m with v_i safe for V_i, an effective demand Δ, and t with

$$((\bigcup_i \underline{\rho}_i)\ f\ \Delta\ v_1\ \ldots\ v_m)\ t = \bot \text{ and } \Delta\ ((\bigcup_i (P_i\ t))\ f\ (V_1\ t)\ \ldots\ (V_m\ t)) \neq \bot \qquad (4)$$

Since Δ as well as function application are continuous functions and $(P_i\ t)$ is a chain, there must be a $k \in \mathbb{N}$ with $\Delta\ (P_k\ t\ f\ (V_1\ t)\ \ldots\ (V_m\ t)) \neq \bot$. Because $\underline{\rho}_k\ f$ is safe for $\lambda t.\ P_k\ t\ f$ by assumption, it follows, that $(\underline{\rho}_k\ f\ \Delta\ v_1\ \ldots\ v_m)\ t \neq \bot$ and finally $((\bigcup_i \underline{\rho}_i)\ f\ \Delta\ v_1\ \ldots\ v_m)\ t \neq \bot$, since $(\underline{\rho}_i)$ is a chain The latter is a contradiction to (4), hence $\bigcup_i \underline{\rho}_i$ is safe for $\lambda t . \bigcup_i (P_i t)$.
q.e.d.

Corollary 4.8 (Safety of the module semantics)
If $m = [\![tds; f_1 = e_1; \ldots; f_n = e_n]\!]$ is a core language module, ρ an import environment and $\underline{\rho}$ an environment of demand propagators, which is safe for ρ. If further for all i, $1 \leq i \leq n$, it is confessed, that safety of $\underline{\rho}$ for ρ implies the safety of $\underline{E}[\![e_i]\!]\ \underline{\rho}$ for $E[\![e_i]\!]\ \rho$, then $\underline{M}[\![m]\!]\ \underline{\rho}$ is safe for $M[\![m]\!]\ \rho$.

Proof. It is well-known, that the minimal fixpoint of a continuous function between c.p.o. can be constructed by an iterative chain.

$\underline{M}[\![m]\!]\ \underline{\rho} = \bigcup_i \underline{\rho}_i$, with $\underline{\rho}_0 = \bot$ and $\underline{\rho}_{i+1} = \rho[\underline{E}[\![e_1]\!]\ \underline{\rho}_i / f_1, \ldots, \underline{E}[\![e_n]\!]\ \underline{\rho}_i / f_n]$
and $M[\![m]\!]\ \rho = \bigcup_i \rho_i$, with $\rho_0 = \bot$ and $\rho_{i+1} = \rho[E[\![e_1]\!]\ \rho_i / f_1, \ldots, E[\![e_n]\!]\ \rho_i / f_n]$.

(ρ_i) and $(\underline{\rho}_i)$ are chains and for all i, $\underline{\rho}_i$ is safe for ρ_i by lemma 4.6 and induction on i. Hence from lemma 4.7, it follows that $\underline{M}[\![m]\!]\ \underline{\rho}$ is safe for $M[\![m]\!]\ \rho$.
q.e.d.

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

Theorem 4.9 (Safety Theorem for the Demand Propagation Semantics)
If m is a core language module, ρ an import environment and $\underline{\rho}$ an environment of demand propagators, which is safe for ρ. Then $\underline{M}[\![m]\!]\ \underline{\rho}$ is safe for $M[\![m]\!]\ \rho$.

Proof. We prove by induction on the structure of e, that if $\underline{\rho}$ is safe for P and e is an arbitrary expression then $\underline{E}[\![e]\!]\ \underline{\rho}$ is safe for $\lambda t.\ E[\![e]\!]\ (P\ t)$. The proposition of the theorem then follows immediately by setting P=const ρ and applying corollary 4.8. We now start the induction on the structure of e:

Let $\underline{\rho}$ be safe for P. In each case let Δ be an effective context demand, $0 \le m \le \mathit{arity}(e)$, v_i safe for V_i for all $1 \le i \le m$, and $\Delta_m = (\underline{E}[\![e]\!]\ \underline{\rho}\ \Delta\ v_1 \ldots v_m)\blacklozenge$.

case e=x or e=C
It is $\underline{E}[\![x]\!]\ \underline{\rho} = \underline{\rho}\ x$ safe for $\lambda t.\ P\ t\ x = \lambda t.\ E[\![x]\!]\ (P\ t)$ by assumption. The same holds for constructor functions.

case $e=e_1\ e_2$
It is $\underline{E}[\![e_1\ e_2]\!]\ \underline{\rho}\ \Delta = \underline{E}[\![e_1]\!]\ \underline{\rho}\ \Delta\ (\underline{E}[\![\ e_2]\!]\ \underline{\rho})$ and $E[\![e_1\ e_2]\!]\ (P\ t) = E[\![e_1]\!]\ (P\ t)\ (E[\![\ e_2]\!]\ (P\ t))$, hence $\Delta_m = (\underline{E}[\![e_1]\!]\ \underline{\rho}\ \Delta\ (\underline{E}[\![e_2]\!]\ \underline{\rho})\ v_1 \ldots v_m)\blacklozenge$. By induction hypothesis is $\underline{E}[\![e_1]\!]\ \underline{\rho}$ safe for $\lambda t.\ E[\![e_1]\!]\ (P\ t)$ as well as $\underline{E}[\![e_2]\!]\ \underline{\rho}$ for $\lambda t.\ E[\![e_2]\!]\ (P\ t)$. Thus

$$(\underline{E}[\![e_1]\!]\ \underline{\rho}\ \Delta\ (\underline{E}[\![e_2]\!]\ \underline{\rho})\ v_1 \ldots v_m)\blacklozenge t = \bot \textit{ implies}$$
$$\bot = \Delta\ (E[\![e_1]\!]\ (P\ t)\ (E[\![e_2]\!]\ (P\ t))\ (V_1\ t) \ldots (V_m\ t)) = E[\![e_1\ e_2]\!]\ (P\ t)\ (V_1\ t) \ldots (V_m\ t)\ .$$

Hence $\underline{E}[\![e_1\ e_2]\!]\ \underline{\rho}$ is safe for $\lambda t.\ E[\![e_1\ e_2]\!]\ (P\ t)$.

case $e=\lambda x.\ e_1$
It is $\underline{E}[\![\lambda x.e_1]\!]\ \underline{\rho}\ \Delta = \underline{\lambda} x.\ \underline{E}[\![e_1]\!]\ \underline{\rho}[x/x]\ \Delta$ and $E[\![\lambda x.e_1]\!]\ (P\ t) = \lambda v.\ E[\![e_1]\!]\ (P\ t)[v/x]$, hence $\Delta_m = ((\underline{\lambda} x.\ \underline{E}[\![e_1]\!]\ \underline{\rho}[x/x]\ \Delta)\ v_1 \ldots v_m)\blacklozenge = (\underline{E}[\![e_1]\!]\ \underline{\rho}[v_1/x]\ \Delta\ v_2 \ldots v_m)\blacklozenge$. It is v_1 safe for V_1, hence $\underline{\rho}[v_1/x]$ safe for $(P\ t)[V_1\ t/x]$ by lemma 4.6 (1). By induction hypothesis $\underline{E}[\![e_1]\!]\ \underline{\rho}[v_1/x]$ is safe for $\lambda t.\ E[\![e]\!]\ (P\ t)[V_1\ t/x]$. Thus

$$(\underline{E}[\![e_1]\!]\ \underline{\rho}[v_1/x]\ \Delta\ v_2 \ldots v_m)\blacklozenge t = \bot \textit{ implies}$$
$$\bot = \Delta\ (E[\![e_1]\!]\ (P\ t)[V_1\ t/x]\ (V_2\ t) \ldots (V_m\ t)) = \Delta\ ((\lambda v.\ E[\![e_1]\!]\ (P\ t)[v/x])\ (V_1\ t) \ldots (V_m\ t))$$
$$= \Delta\ (E[\![\lambda x.\ e_1]\!]\ (P\ t)\ (V_1\ t) \ldots (V_m\ t))\ .$$

Hence $\underline{E}[\![\lambda x.e]\!]\ \underline{\rho}$ is safe for $\lambda t.\ E[\![\lambda x.e]\!]\ (P\ t)$.

case e=case e_c of $C_1\ v_{11}\ldots v_{1a_1} \Rightarrow e_1; \ldots; C_n\ v_{n1}\ldots v_{na_n} \Rightarrow e_n; v \Rightarrow e_{def}$

It is $\Delta_m = (\underline{E}[\![e_c]\!]\ \underline{\rho}\ \mathsf{WHNF}\ \&\ (\pi_1\ v_1\ \ldots\ v_m \mid \ldots \mid \pi_n\ v_1\ \ldots\ v_m \mid \pi_{def}\ v_1\ \ldots\ v_m))\blacklozenge$, since $\mathit{lift}^{type(e)}(\underline{E}[\![e_c]\!]\ \underline{\rho}\ \mathsf{WHNF})\ v_1\ \ldots\ v_m = \underline{E}[\![e_c]\!]\ \underline{\rho}\ \mathsf{WHNF}$, with the definitions of π_i and π_{def} given in the demand propagation semantics.

Assume now $\Delta_m\ t = \bot$. From the definition of & it follows that

$$(\underline{E}[\![e_c]\!]\ \underline{\rho}\ \mathsf{WHNF})\ t = \bot \text{ or} \tag{5}$$

$$(\pi_1\ v_1\ \ldots\ v_m \mid \ldots \mid \pi_n\ v_1\ \ldots\ v_m \mid \pi_{def}\ v_1\ \ldots\ v_m)\ t = \bot \tag{6}$$

1. Consider $(\underline{E}[\![e_c]\!]\ \underline{\rho}\ \mathsf{WHNF})\ t = \bot$. By induction hypothesis is $\underline{E}[\![e_c]\!]\ \underline{\rho}$ safe for $\lambda t.\ E[\![e_c]\!]\ (P\ t)$, hence $\mathsf{WHNF}\ (E[\![e_c]\!]\ (P\ t)) = \bot$, which is equivalent to $E[\![e_c]\!]\ (P\ t) = \bot$. From the standard semantics of the case expression then follows $E[\![e]\!]\ (P\ t) = \bot$, hence $\Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t)\ \ldots\ (V_m\ t)) = \bot$ for arbitrary V_i, since Δ is effective.
2. In the second case we consider that (6) is true but (5) is false. Then define $E = \lambda v.\ E[\![e_c]\!]\ (P\ v)$. We now have to look at the standard semantics of the case-expression, which relies on the value of E t:

   (a) If $E\ t = \bot$, then also $E[\![e]\!]\ (P\ t)\ (V_1\ t)\ \ldots\ (V_m\ t) = \bot$, regardless of the values of the arguments, hence also $\Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t)\ \ldots\ (V_m\ t)) = \bot$.

   (b) If $tag(E\ t) = C_k$ for one k, $1 \le k \le n$, then $E[\![e]\!]\ (P\ t) = E[\![e_k]\!]\ (P'\ t)$ with $P'\ t = (P\ t)[proj_{C_k,1}(E\ t)/v_{k1}, \ldots, proj_{C_k,a_k}(E\ t)/v_{ka_k}]$. From (6) follows:

$$(\pi_k\ v_1\ \ldots\ v_m)\blacklozenge\ t = \bot \textit{ with}$$
$$\pi_k = \mathit{lift}^{type(e)}(\underline{E}[\![e_c]\!]\ \underline{\rho}\ (C_k\ \mathsf{NO}\ldots\mathsf{NO}))\ \&\ \underline{E}[\![e_k]\!]\ \underline{\rho}[w_1/v_{k1}, \ldots, w_{a_k}/v_{ka_k}]\ \Delta \textit{ and}$$
$$w_j = \lambda\Delta.\ \underline{E}[\![e_c]\!]\ \underline{\rho}\ (C_k\ \mathsf{NO}\ldots\Delta^{(j)}\ldots\mathsf{NO})$$

   $\underline{\rho}[w_1/v_{k1}, \ldots, w_{a_k}/v_{ka_k}]$ is safe for $\lambda t.\ (P'\ t)$ by lemmata 4.5 (1) and 4.6 (2), hence we can use the induction hypothesis for both subterms, yielding:

$$(C_k\ \mathsf{NO}\ldots\mathsf{NO})\ (E[\![e_c]\!]\ (P\ t)) = \bot \textit{ or}$$
$$\bot = \Delta\ (E[\![e_k]\!]\ (P'\ t)\ (V_1\ t)\ \ldots\ (V_m\ t)) = \Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t)\ \ldots\ (V_m\ t)).$$

   The definition of $C_k\ \mathsf{NO}\ldots\mathsf{NO}$ shows, that the first term is always false, if $tag(E[\![e_c]\!]\ (P\ t)) = C_k$, hence $\Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t)\ \ldots\ (V_m\ t)) = \bot$.

(c) If $\bot \neq tag(E\ t) \notin CS=\{C_1,\ldots,C_n\}$, then $E[\![e]\!]\ (P\ t) = E[\![e_{def}]\!]\ (P'\ t)$ with $(P'\ t) = (P\ t)[rest_{CS}(E\ t)/v]$. From (6) follows:

$$(\pi_{def}\ v_1 \ldots v_m)\blacklozenge t = \bot \textit{ with}$$
$$\pi_{def} = lift^{type(e)}(\underline{E}[\![e_c]\!]\ \underline{\rho}\ (WHNF\backslash CS)) \ \&\ \underline{E}[\![e_{def}]\!]\ \underline{\rho}[(\lambda\Delta.\ \underline{E}[\![e_c]\!]\ \rho\ \Delta\backslash CS)/v]\ \Delta$$

$\underline{\rho}[(\lambda\Delta.\ \underline{E}[\![e_c]\!]\ \rho\ \Delta\backslash CS)/v]$ is safe for $\lambda t.\ P'\ t$ by lemmata 4.5 (2) and 4.6 (1), hence we can use the induction hypothesis for both subterms, yielding:

$$(WHNF\backslash CS)\ (E[\![e_c]\!]\ (P\ t)) = \bot \textit{ or}$$
$$\bot = \Delta\ (E[\![e_{def}]\!]\ (P'\ t)\ (V_1\ t) \ldots (V_m\ t)) = \Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t) \ldots (V_m\ t)).$$

The first term can only be true, if $E[\![e_c]\!]\ (P\ t) = \bot$ or $tag(E\ t) \in CS$, both excluded in this case, hence $\Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t) \ldots (V_m\ t)) = \bot$.

Thus in all cases $\Delta_m\ t = \bot$ implies $\Delta\ (E[\![e]\!]\ (P\ t)\ (V_1\ t) \ldots (V_m\ t)) = \bot$.

case e=let m in e'

It is $\underline{E}[\![e]\!]\ \underline{\rho} = \underline{E}[\![e']\!]\ (\underline{M}[\![m]\!]\ \underline{\rho})$ and $E[\![e]\!]\ (P\ t) = E[\![e']\!]\ (M[\![m]\!]\ (P\ t))$.

From the induction hypothesis (applying to all definitions in m) and lemma 4.7 it follows that $\underline{M}[\![m]\!]\ \underline{\rho}$ is safe for $\lambda t.\ M[\![m]\!]\ (P\ t)$, and hence $\underline{E}[\![e']\!]\ (\underline{M}[\![m]\!]\ \underline{\rho})$ is safe for $\lambda t.\ E[\![e']\!]\ (M[\![m]\!]\ (P\ t))$ again by induction hypothesis.

q.e.d.

5 Conclusions and Further Work

Higher order demand propagation semantics is a non-standard semantics for functional languages, which is able to capture generalised strictness properties for higher order polymorphic functions. The examples in [Pa98] showed how demand propagators can be used to find very accurate generalised strictness. This report provides a formal proof for the correctness of this strictness information.

On the basis of this we plan to design a strictness analysis, which can be used in a compiler to hint for optimisation possibilities. A raw idea for this design has also been presented in [Pa98], and a prototype of the analysis is in implementation.

References

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

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

[Pa98] D. Pape: *Higher Order Demand Propagation.* Technical Report, Dept. of Comp. Science, Freie Universität Berlin 1998. Available at <http://www.inf.fu-berlin.de/~pape/papers/>

[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/>