Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/36506

Please be advised that this information was generated on 2019-11-11 and may be subject to change.

STRICTNESS ANALYSIS VIA RESOURCE TYPING

ERIK BARENDSEN AND SJAAK SMETSERS

Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
e-mail address: E.Barendsen@cs.ru.nl

Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
e-mail address: S.Smetsers@cs.ru.nl

Abstract. We present a new typing system for strictness analysis of functional programs. The system extends standard typing (including recursive data types) with strictness annotations and subtyping. Strictness typing is shown to be sound with respect to a natural operational semantics. We demonstrate that strictness types can be computed effectively.

Introduction

Static analysis techniques are crucial components of state-of-the-art implementations of lazy functional programming languages. These techniques are employed to determine properties of the specified functions, as a service to the programmer (for instance to detect inconsistencies), but also to provide for the generation of efficient executables by a compiler.

Strictness analysis is meant to determine dependencies between evaluation of arguments of functions and evaluation of their results. Strictness information can be used to change inefficient *call-by-need* evaluation into efficient *call-by-value* evaluation. This gain in efficiency lies in the fact that construction of unevaluated expressions (so-called *closures*) is circumvented.

Traditional strictness analysers are based on *abstract interpretation* using a non-standard semantics to derive strictness properties, see [Myc81], [BHA85] and [Wad87]. This involves determination of fixed points which in general cannot be implemented efficiently, thus making the method unsuitable for compilers.

Another technique is *abstract reduction*, which analyses evaluation by mimicking reduction on sets of concrete values extended with generic elements for undefinedness. This technique approximates ordinary computations much closer than for instance abstract interpretation or strictness typing. The rewriting semantics is adjusted by specifying the behaviour of functions on non-standard elements. Abstract reduction sequences may not

2000 ACM Subject Classification: 68N18, 68Q55, 03B15 (68N18).

Key words and phrases: strictness analysis, functional programming languages, typing, operational semantics, recursive types.

REFLECTIONS ON TYPE THEORY, λ-CALCULUS, AND THE MIND — Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday

Copyright © 2007 by Erik Barendsen and Sjaak Smetsers

terminate. A special technique called *reduction path analysis* is used to cut off these sequences in a way that does keep most of the strictness information intact; see [Nöc93].

The present paper presents a type-theoretic approach to strictness analysis. Our method is based on standard Hindley-Milner typing systems. The idea is to annotate the types of function arguments with so-called *strictness attributes*. With these annotations we are able to determine how the arguments are used inside the function body. For our usage analysis we regard inputs as rescources and monitor them via a so-called *resource conscious* typing system. This idea has also been used in the context of reference analysis, for example with linear typing (based on linear logic, [Wad90]) and uniqueness typing [BS96]. We will use a principle similar to uniqueness typing: resource properties are encoded as (polymorpic) type attributes and different uses of these resources are combined via special typing rules and a subtyping relation.

Compared to traditional strictness analysers, our strictness typing has some important advantages. First, strictness typing can be combined with ordinary typing; no additional analysis phase is need during compilation. Second, by adopting the Hindley-Milner approach for recursion, we can avoid fixed point computations, making the algorithm much more efficient. Moreover, the strictness typing system is able to deal with arbitrary (recursive and non-recursive) data-types as well as with higher order functions. Apart from the outermost type level, strictness attributes are added to each internal type component. Thus, strictness information is not restricted to traditional head-normal form evaluation only. In this respect it differs from other approaches such as [CDG02].

1. Syntax

We will focus on evaluation in functional programming languages by describing a 'core language' in which expressions are built up from applications of function symbols and data constructors, all with a fixed arity. Pattern matching is expressed by a construction $\textsc{case} \cdots \textsc{of} \cdots$.

Definition 1.1. (i) The set of *expressions* is defined by the following syntax. Below, x ranges over variables, C over constructors and f over function symbols.

$$\begin{array}{lcl} E & ::= & x \mid S(E_1, \ldots, E_k) \mid \textsc{case}\ E\ \textsc{of}\ P_1 \rightarrow E_1 \cdots P_n \rightarrow E_n \\ S & ::= & \mathsf{C} \mid f \\ P & ::= & \mathsf{C}(x_1, \ldots, x_k) \end{array}$$

(ii) The set of free variables (in the obvious sense) of E is denoted by $\mathrm{FV}(E)$.

(iii) A *function definition* is an expression of the form

$$f(x_1, \ldots, x_k) = E$$

where all the x_i's are disjoint and $\mathrm{FV}(E) \subseteq \{x_1, \ldots, x_k\}$.

In the sequel we will assume that a fixed set of function definitions (a 'functional program') is given.

Example 1.2. A function that computes the size of a tree can be defined as follows.

$$\begin{array}{lllll} \mathit{size}(t) = \textsc{case}\ t\ \textsc{of} & \mathsf{Leaf} & \rightarrow & 0 \\ & \mathsf{Node}(x, l, r) & \rightarrow & \mathit{plus}(\mathit{plus}(\mathit{size}(l), \mathit{size}(r)), 1) \end{array}$$

2. Semantics

In this section we will characterize the process of (lazy) evaluation of expressions to data objects. We focus on evaluation to an expression starting with a data constructor, a so-called *constructor normal form* (*cnf* for short). The actual computation of the resulting data object is done via iterated evaluation to constructor normal form.

We will describe a straightforward evaluation in which computations are done by successive substitutions or replacements (*call-by-name*). With some adjustments to the syntax it is possible to incorporate proper sharing, including cycles, with a *call-by-reference* semantics in the style of [Lau93], see [BS99]. Since this extension is not essential for studying strictness analysis, we will describe the simple system without sharing.

Substitution semantics. For a clear presentation we limit the set of constructors to $\Box$, the binary constructor $\langle\ ,\ \rangle$ (intended meaning: pairing) and to the unary constructors inl and inr (intended meaning: injections of disjoint union). The $\Box$ constructor is used as a representation of all elementary values; the other constructors represent compound values. To prepare for a typed variant, we assume that each CASE construct handles either the product constructor or the sum constructors.

Definition 2.1. (i) An expression is in *constructor normal form* if it has one of the following shapes.

$$\Box, \langle E, E' \rangle, \mathsf{inl}\ E, \mathsf{inr}\ E$$

(ii) Let E, V be expressions. Then E is said to *evaluate to* V (notation $E \Downarrow V$) if $E \Downarrow V$ can be produced in the following derivation system.

$$x \Downarrow x \qquad \mathsf{C}(\vec{E}) \Downarrow \mathsf{C}(\vec{E})$$

$$\frac{E[\vec{x} := \vec{E}] \Downarrow V}{f(\vec{E}) \Downarrow V} \quad \text{if } f(\vec{x}) = E$$

$$\frac{E \Downarrow \mathsf{C}_j(\vec{E}) \qquad E_j[\vec{x} := \vec{E}] \Downarrow V}{\text{CASE } E \text{ OF } \ldots (\mathsf{C}_j(\vec{x}) \to E_j) \ldots \Downarrow V}$$

Strictness. We will introduce a notion of *strictness* for expressions and functions.

Definition 2.2. (i) Let E be an expression. E is *defined* (notation $E\Downarrow$) if $E \Downarrow V$ for some V. Otherwise E is *undefined* (notation $E\Uparrow$).

(ii) E is *strict in* x if for all expressions A

$$A\Uparrow \ \Rightarrow\ E[x := A]\Uparrow.$$

(iii) A unary function f (say with definition $f(x) = E$) is *strict* if E is strict in x. (Likewise one formulates argumentwise strictness of functions of larger arity.)

Example 2.3. One can easily check that the boolean function *and* is strict in its first and not strict (lazy) in its second argument.

$$\begin{array}{lrcl} \mathit{and}(x, y) = \text{CASE } x \text{ OF} & \mathsf{True} & \to & y \\ & \mathsf{False} & \to & \mathsf{False} \end{array}$$

The usage of strictness information in evaluation of functional programs is based on the following observation. Whenever a function f is strict, one could modify the evaluation of f-applications by the call-by-value rule

$$\frac{E' \Downarrow V' \qquad E[x := V'] \Downarrow V}{f(E') \Downarrow V} \quad \text{if } f(x) = E$$

without affecting definedness ('termination') of graph expressions. By evaluating function arguments before evaluating the function body one avoids duplicating unevaluated expressions in the case of copying semantics.

Our aim is to refine the notion of strictness to evaluation properties of various parts of compound data objects (instead of just the outermost constructor, like above) and to determine strictness properties in an effective way.

3. Typing

In programming languages, typing systems are used to ensure consistency of function applications: the type of each function argument should match some specified input type. In functional languages, functions are usually regarded as *polymorphic*: their types are given in a schematic form. The most common typing systems for these languages are based on Hindley/Milner typing. An important property of these standard typing systems is the so-called *principal type property*: for every typable expression E there is exists a *principal typing*, i.e. a type representing all possible types for E. These principal types can be computed effectively, see the next section.

Such standard type systems can be extended to characterize properties such as reference counts and strictness. This is done by annotating the type expressions. A common aspect of these *non-standard type systems* is that they contain a refined analysis of *resources*, i.e. occurrences of input objects (the free variables in our expressions).

We will capture evaluation properties by an extension of standard typing.

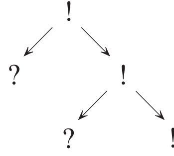
3.1. **Evaluation structures.**

Our data objects are built up from data *constructors*. Data objects can be complex or even infinite (in the case of lazy evaluation). The part of the data object that is actually used ('needed') depends on the context of the object. When a list is used by a function selecting its third element, only the spine structure up to depth 3 and the third element itself are needed to produce the result.

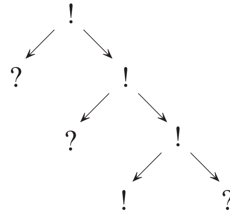
The result of a function may also be a compound data object. It therefore makes sense to analyse dependencies between the *evaluation structure* of function results and the evaluation structure of arguments.

We can represent evaluation structures by trees corresponding to the syntax of data objects. We indicate evaluation by '!' and non-evaluation by '?'. For tuples (pairs), for example, we can represent 'not evaluated', 'evaluated up to the Pair constructor' and 'evaluation of the first element' respectively by the trees

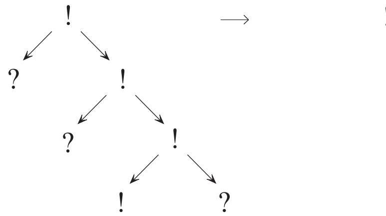
For tuples of tuples, the following tree represents evaluation of the second element of the second element.



The evaluation structure for the lists in the example of the previous paragraph is represented by

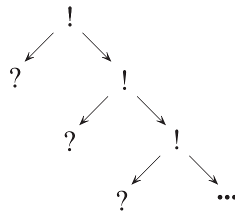


Evaluation properties of functions can be expressed by evaluation structures for result and inputs. Strictness properties can be derived from these structures. Our 'third element selection' is strict with respect to the first three spine components and the third element: if either of these is undefined (or better: cannot be evaluated) the result is undefined.



One could say that the '!' of the result is connected to ('caused by') the '!'occurrences in the argument structure. Our goal is to analyze these connections. For compound result structures, we will be able to express these connections componentwise using 'evaluation variables', see 4.2.

We will use a typing system for the evaluation analysis. Therefore we restrict the evaluation structures to those that are expressible by *annotating type expressions*. E.g., the annotated type $(\alpha^! \times \beta^?)^!$ corresponds to the third tuple evaluation structure above. A possible annotation for lists over some type α is $\mathsf{List}^!(\alpha^?)$, corresponding to the (infinite) structure



3.2. **Strictness typing.** For strictness analysis via typing, we will annotate types and typing statements with *strictness attributes*. Each typing statement expresses the relation between (cnf) the evaluation structures of the parameters (the free variables) and of the result of an expression. Strictness of E in x corresponds to a typing statement

$$x{:}\sigma^! \vdash E : \tau^!,$$

which can be read as 'if x is undefined then E is undefined' or equivalently 'if E has a cnf then x has a cnf'.

Strictness annotations are not restricted to the outermost level. For example,

$$p{:}(\sigma^! \times \tau^?)^! \vdash \text{CASE OF } \langle x, y\rangle \to x : \sigma^!.$$

Definition 3.1. (i) The collection of *strictness types* is obtained by annotating standard types. Below, α ranges over type variables, and u over the attributes ! and ?.

$$S \ ::= \ \alpha^u \mid (S_1 \times S_2)^u \mid (S_1 + S_2)^u \mid \mathbf{1}^u$$

(ii) The outermost attribute of S is denoted by $\ulcorner S \urcorner$.

(iii) We define an ordering on strictness types with the same underlying standard type (i.e., the type obtained by omitting all attributes). This is done via the ordering

$$! \le ?$$

on attributes. E.g.

$$(\sigma^v \times \tau^w)^u \le (\sigma^{v'} \times \tau^{w'})^{u'} \quad \text{iff} \quad u \le u', v \le v', w \le w'.$$

(iv) Let S, S' be strictness types with the same underlying standard type. Then $S \sqcap S'$ denotes the type obtained by componentwise taking the minimum of the attributes in S and S'.

The function symbols are supplied with a (strictness) type scheme by a *function type environment* $\mathcal{F}$, containing declarations of the form

$$f : (S_1, \ldots, S_k) \to T,$$

where k is the arity of f. Specific instances of this scheme can be obtained via *substitutions* replacing type variables with strictness types. In the type scheme itself type variables will be annotated. These annotations are not affected by the substitution, only the variables themselves are replaced. This means that the outermost attributes of the substituted strictness types are in fact ignored. We write

$$\mathcal{F} \vdash f : (S'_1, \ldots, S'_k) \to T'$$

if there is a substitution $*$ such that $S'_1 = S^*_1, \ldots, S'_k = S^*_k, T' = T^*$.

Definition 3.2. Derivation of strictness typing statements $B \vdash E : S$ is characterized by the following rules.

$$x{:}S \vdash x : S \quad \text{(Var)}$$

$$\frac{\mathcal{F} \vdash f : \vec{S} \to T \qquad B_i \vdash E_i : S_i}{\vec{B} \vdash f\vec{E} : T} \text{(App)}$$

$$\frac{B \vdash E : S \qquad B' \vdash E' : T \qquad u \leq \ulcorner S \urcorner \qquad u \leq \ulcorner T \urcorner}{B, B' \vdash \langle E, E' \rangle : (S \times T)^u} (\times\text{-I})$$

$$\frac{B \vdash E : (S \times T)^u \qquad B', x{:}S, y{:}T \vdash E' : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \textsc{case}\ E\ \textsc{of}\ \langle x, y \rangle \to E' : R}$$

$$\frac{B \vdash E : S \qquad u \leq \ulcorner S \urcorner}{B \vdash \mathsf{inl}\ E : (S + T)^u} (+\text{-I-l}) \qquad \frac{B \vdash E : T \qquad u \leq \ulcorner T \urcorner}{B \vdash \mathsf{inr}\ E : (S + T)^u} (+\text{-I-r})$$

$$\frac{B \vdash E : (S + T)^u \qquad B', x{:}S \vdash P : R \qquad B', y{:}T \vdash Q : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \textsc{case}\ E\ \textsc{of}\ (\mathsf{inl}\ x \to P)(\mathsf{inr}\ y \to Q) : R} (+\text{-E})$$

$$\frac{B \vdash E : S \qquad T \leq S}{B \vdash E : T} \text{(Sub)} \qquad \frac{B \vdash E : S}{B, x{:}\tau^? \vdash E : S} \text{(Wea)} \qquad \frac{B, x{:}S, y{:}T \vdash E : U}{B, z{:}S \sqcap T \vdash E[x := z, y := z] : U} \text{(Con)}$$

In order to express the soundness of the system we refine the notion of definedness (see Definition 2.2 (i)) by parameterizing the operational semantics by strictness types.

Definition 3.3. *Typed evaluation* is defined as follows.

(i) For each S, the notion $E \Downarrow_S V$ (E S-evaluates to V) is inductively defined by the following rules.

$$E \Downarrow_{\sigma^?} E \qquad \Box \Downarrow_{\mathbf{1}^!} \Box$$

$$\frac{E \Downarrow \langle E_1, E_2 \rangle \qquad E_1 \Downarrow_S V_1 \qquad E_2 \Downarrow_T V_2}{E \Downarrow_{(S \times T)^!} \langle V_1, V_2 \rangle} \qquad \frac{E \Downarrow \mathsf{inl} E' \qquad E' \Downarrow_S V}{E \Downarrow_{(S+T)^!} \mathsf{inl} V} \qquad \frac{E \Downarrow \mathsf{inr} E' \qquad E' \Downarrow_T V}{E \Downarrow_{(S+T)^!} \mathsf{inr} V}$$

(ii) We say that E is S-*defined* if $E \Downarrow_S V$ for some V. Otherwise E is S-*undefined* (notation $E {\Uparrow_S}$).

The notion of strictness can now be refined. An expression E, say with one free variable x, is S-*to*-T-*strict* if for every A

$$A {\Uparrow_S} \quad \Rightarrow \quad E[x := A] {\Uparrow_T}.$$

Typing statements can now be interpreted as strictness statements; in fact we have the following soundness property

$$x : S \vdash E : T \ \Rightarrow\ E \text{ is } S\text{-to-}T\text{-strict}.$$

3.3. **Soundness.** In the remainder of this section we will demonstrate that the type system is sound, i.e., typing statements are valid with respect to our characterization of strictness.

Definition 3.4. (i) For each type S, the set of *S-undefined values* (notation $[\![S]\!]$) is defined by

$$[\![S]\!] = \{E \mid E{\Uparrow}_S\}.$$

(ii) Let ρ be a function from variables to expressions. By $[\![E]\!]_\rho$ we denote the result of substituting $\rho(x)$ for x in E, for each free variable x of E.

(iii) We say that ρ *satisfies* $E : T$ (notation $\rho \vDash E : T$) if $[\![E]\!]_\rho \in [\![T]\!]$. Moreover, ρ satisfies B (notation $\rho \vDash B$) if $\rho(x) \in [\![T]\!]$ for some $x : T$ in B.

(iv) B *satisfies* $E : T$ (notation $B \vDash E : T$) if for all ρ

$$\rho \vDash B \quad \Rightarrow \quad \rho \vDash E : T.$$

Theorem 3.5 (Soundness)**.** $B \vdash E : T \quad \Rightarrow \quad B \vDash E : T.$

4. Decidability

In this section we will discuss how strictness variants of traditional typings can be computed.

Standard type inference usually takes some *syntax directed* type system as a starting point. In a syntax directed system, each syntactic construction has exactly one typing rule. Typing an expression E boils down to stepwise reconstruction of a type derivation for E (including type derivations for each subexpression of E). It is convenient to split the type inference into two phases (see [Wan87]): generating requirements in the form of *type equations* and solving these equations by means of some *unification*. The result is called a *principal typing* for E, usually given as a *type scheme* with type variables as placeholders. Thus, such type schemes introduce a form of (weak) *polymorphism*.

Strictness type inference can be done in a similar way. Since strictness typing involves subtyping, one generates type *inequalities* rather than type equations. The transformation of the derivation system into a syntax directed variant is somewhat involved.

4.1. **A syntax directed system.** The system has three rules not exclusively connected to a specific syntactic construction: subsumption, weakening and contraction. We will incorporate the effect of these rules in the other rules.

The effect of the *subsumption* rule is taken into account by adding subtyping constraints to the variable, application and constructor-introduction rules.

We compensate for the *weakening* rule by allowing the bases in the conclusions of the variable, constant and application rules to be extended with an arbitrary set of declarations with type attribute ?.

To deal with the contraction rule, we change our administration of declarations in bases. The base unions B, B' in our system are not longer regarded as disjoint union. We take multiple occurrences into account by using ordinary unions, together with an operation to combine the types of common variables. This operation takes care of 'simultaneous' occurrences of variables. Multiple occurrences in the 'mutually excluding' branches of a case construction are dealt with by introducing a new basis consisting of supertypes of the variable types used in the respective branches.

After formalizing the above auxiliary operations we will be ready to give the syntax directed version of the strictness typing system.

Definition 4.1. (i) With $B^?$ we denote a *lazy basis* with only declarations of the form $x{:}\sigma^?$.

(ii) The *union* of B and B', denoted as B, B', combines $x{:}S \in B$ with $x{:}S' \in B'$ to $x{:}S \sqcap S'$.

(iii) *Subtyping* for bases is defined as follows. $B \leq B'$ if for each $x{:}S' \in B'$ there exists S with $x{:}S \in B$ and $S \leq S'$.

Definition 4.2. The syntax directed system looks as follows.

$$\frac{S \leq S'}{B^?, x{:}S' \vdash x : S}\ \text{(variable)}$$

$$B^? \vdash \Box : \mathbf{1}^u \quad \text{(constant)}$$

$$\frac{\mathcal{F} \vdash f : \vec{S} \rhd T \qquad B_i \vdash E_i : S_i \qquad R \leq T}{B^?, \vec{B} \vdash f\vec{E} : R}\ \text{(application)}$$

$$\frac{B \vdash E : S \qquad B' \vdash E' : T \qquad u \leq \ulcorner S \urcorner \qquad u \leq \ulcorner T \urcorner \qquad R \leq (S \times T)^u}{B, B' \vdash \langle E, E' \rangle : R}\ (\times\text{-I})$$

$$\frac{B \vdash E : (S \times T)^u \qquad B', x{:}S, y{:}T \vdash E' : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \textsc{case}\ E\ \textsc{of}\ \langle x, y \rangle \to E' : R}\ (\times\text{-E})$$

$$\frac{B \vdash E : S \qquad u \leq \ulcorner S \urcorner \qquad R \leq (S + T)^u}{B \vdash \mathsf{inl}\ E : R}\ (+\text{-I-l})$$

$$\frac{B \vdash E : T \qquad u \leq \ulcorner T \urcorner \qquad R \leq (S + T)^u}{B \vdash \mathsf{inr}\ E : R}\ (+\text{-I-r})$$

$$\frac{B \vdash E : (S+T)^u \qquad \begin{array}{c} B', x{:}S \vdash P : R \\ B'', y{:}T \vdash Q : R \end{array} \qquad \ulcorner R \urcorner \leq u \qquad \begin{array}{ccc} B' & \leq & B''' \\ B'' & \leq & B''' \end{array}}{B, B''' \vdash \textsc{case}\ E\ \textsc{of}\ (\mathsf{inl}\ x \to P)(\mathsf{inr}\ y \to Q) : R}\ (+\text{-E})$$

One can show that the above system is equivalent to the original version, i.e., they have the same set of provable statements.

4.2. **Strictness polymorphism.** The type schemes in the strictness system will involve attribute variables besides the standard type variables. Since subtyping depends on inequalities between type attributes, our type schemes will contain dependencies between attributes. We will express these dependencies as (finite) sets of attribute inequalities called *attribute environments*. The following example illustrates the use of these environments. The possible strictness types of the function *fst*, defined by

$$\mathit{fst}(p) = \textsc{case}\ p\ \textsc{of}\ \langle x, y \rangle \to x,$$

can be expressed by the strictness type scheme

$$fst : (\alpha^a \times \beta^?)^c \to \alpha^a \;\mid\; a \le c.$$

The attribute environment denotes restrictions on instantiations. For example, $a := !, c := !$ and $a := !, c := ?$ are valid instantiations, but $a := ?, c := !$ is not, since $? \not\le !$.

When solving attribute inequalities (viz. in the case of union of bases), one needs the minimum (or rather the greatest lower bound) of strictness attributes. In the resulting type schemes, this operation is indicated with $u \sqcap v$.

In this short paper we will refrain from a formal treatment of attribute environments. Instead, we give an example. The function

$$double(x) = \langle x, x \rangle$$

can be typed with

$$double : \alpha^a \to (\alpha^b \times \alpha^c)^d \;\mid\; d \le b, d \le c, \quad b \sqcap c \le a.$$

To illustrate the power of the resulting system, consider the combination of the above operation with a projection:

$$x{:}\alpha^a \vdash fst(double(x)) : \alpha^b \;\mid\; b \le a.$$

Our approach gives more refined results than traditional methods. If one only keeps track of plain cnf strictness, the dependency between evaluation of $fst(double(x))$ and of x would not be visible.

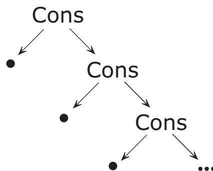
The actual typing algorithm roughly works as follows. First, the syntax directed system is used to generate strictness type inequalities for a given expression. These inequalities are then regarded as a set of standard type equations, by ignoring all strictness attributes and by considering type inequalities and type intersections as equations. These equations are solved using unification, thus giving a solution of the 'standard part'. This solution is substituted in the original strictness type inequalities, lifting the substituted standard types to strictness types using fresh attribute variables. The resulting inequalities can be translated into a set of attribute inequalities, which can be solved and simplified. The result is a principal strictness typing for the original expression.

Theorem 4.3 (Principal Strictness Typing Theorem)**.** *Principal strictness typings can be computed effectively.*

5. Recursive data structures

Thus far we have only considered non-recursive data structures. In this section we will describe an extension of the theory to lists. This example data structure will be paradigmatic for all recursive data types.

Lists are built up from constructors Nil (the empty list) and Cons (constructing a list from a head element and a list). A list is typically of the form



where the sequence of Cons constructors is called the *spine* of the list. By $\mathsf{List}(\sigma)$ we denote the type of lists of σ objects.

There are several possibilities for handling strictness information in lists. We start with the simplest one. Observing the recursive structure of lists, cnf-evaluation is interpreted as evaluation of the outermost constructor and (iterated) cnf-evaluation of the tail of the list. Thus, cnf-evaluation will result in the complete evaluation of the spine.

$$\mathsf{Nil} \Downarrow_{\mathsf{List}^!(S)} \mathsf{Nil} \qquad \frac{E \Downarrow \mathsf{Cons}(E_1, E_2) \qquad E_1 \Downarrow_S V_1 \qquad E_2 \Downarrow_{\mathsf{List}^!(S)} V_2}{E \Downarrow_{\mathsf{List}^!(S)} \mathsf{Cons}(V_1, V_2)}$$

The typing rules reflect this uniform treatment.

$$\vdash \mathsf{Nil} : \mathsf{List}^u(S)\ (\text{List-I-Nil}) \qquad \frac{B \vdash E : S \qquad B' \vdash L : \mathsf{List}^u(S) \qquad u \leq \ulcorner S \urcorner}{B, B' \vdash \mathsf{Cons}(E, L) : \mathsf{List}^u(S)}\ (\text{List-I-Cons})$$

$$\frac{B \vdash E : \mathsf{List}^u(S) \qquad B' \vdash P : R \qquad B', x{:}S, \ell{:}\mathsf{List}^u(S) \vdash Q : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \textsc{case}\ E\ \textsc{of}\ (\mathsf{Nil} \to P)(\mathsf{Cons}(x, \ell) \to Q) : R}\ (\text{List-E})$$

Example 5.1.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{reverse} : (\mathsf{List}^b(\alpha^a), \mathsf{List}^b(\alpha^a)) \to \mathsf{List}^b(\alpha^a)} \\ \mathit{reverse}(l, a) = \textsc{case}\ l\ \textsc{of} & \mathsf{Nil} & \to\ a \\ & \mathsf{Cons}(h, t) & \to\ \mathit{reverse}(t, \mathsf{Cons}(h, a)) \end{array}$$

In some cases, it is useful to distinguish between plain and iterative cnf evaluation, i.e., between evaluation of the topmost constructor and evaluation of the spine. A very simple example is the function hd that takes the head element of a list.

$$hd(l) = \textsc{case}\ l\ \textsc{of} \quad \mathsf{Cons}(h, t) \quad \to \quad h$$

A straightforward strictness type would be

$$hd : \mathsf{List}(\alpha^?)^a \to \alpha^b \ \mid \ b \leq a$$

indicating that the argument of hd will be evaluated if its result is needed. However, annotating the argument as strict in our extended system would wrongly indicate that the evaluation of the whole spine is required to determine the result of hd. Hence we are forced to type the function conservatively as

$$hd : \mathsf{List}^?(\alpha^?) \to \alpha^b$$

This problem can be solved by extending the list types with an extra attribute that enables us to distinguish between plain cnf and (complete) spine evaluation. Of course, this also requires a modification of the evaluation rules. We will not go into these details, but only illustrate the expressive power of such an extension with an example.

First, hd can now be typed as follows.

$$hd : \mathsf{List}^?(\alpha^?)^a \to \alpha^b \ \mid \ b \leq a$$

The outermost attribute a indicates that the argument of hd is evaluated whenever its result is needed (in that case the attribute b becomes ! and hence it is allowed to choose ! for a as

well). And also *reverse* can be typed more accurately.

$$reverse : (\mathsf{List}^c(\alpha^a)^c, \mathsf{List}^b(\alpha^a)^b) \to \mathsf{List}^b(\alpha^a)^c$$

In contrast to the previous type of *reverse* the difference between the first and the second argument becomes apparent: Even simple cnf-evaluation of an application of *reverse* will result in the complete evaluation of the spine of the first argument (the plain and spine attribute of this argument are the same). For the second argument this is not the case. This argument will only be evaluated when the whole spine of the *reverse*'s result is needed. How strictness properties of functions are combined is illustrated by the following function *last* that takes the last element of a list.

$$last(l) \quad = \quad hd(reverse(l, \mathsf{Nil}))$$

Of course, to get to the last element, the whole spine has to be traversed. This evaluation property is reflected by the derived strictness type of this function.

$$last : \mathsf{List}^b(\alpha^a)^b \to \alpha^c \;\mid\; c \leq b$$

It is clear that adding more attributes gives a refinement of characterizations of evaluation properties. For instance, in the *hd* function the elements of a list are treated uniformly. One could think of an extra attribute corresponding to the first element (and the original attribute to all other elements respectively). In that way we are able to specify that *hd* not only evaluates the first constructor of the spine but also the first element.

References

[BHA85] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Proc. of Workshop on Programs as Data Objects*, pages 42–62. DIKU, Denmark, Springer Verlag, LNCS 217, 1985.

[BS96] E. Barendsen and J.E.W. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[BS99] E. Barendsen and J.E.W. Smetsers. Graph rewriting aspects of functional programming. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 63–102. World Scientific, 1999.

[CDG02] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard-type inference. *Theoretical Computer Science*, 272:69–112, 2002.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation,. In *Proc. of POPL'93: Twentieth annual ACM symposium on Principles of Programming Languages*, pages 144–154. Charleston, South Carolina, 1993.

[Myc81] A Mycroft. *Abstract interpretation and optimising transformations for applicative programs.* PhD thesis, University of Edinburgh, 1981.

[Nöc93] E.G.J.M.H. Nöcker. Strictness analysis using abstract reduction. In *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 255–266. Kopenhagen, ACM Press, 1993.

[Wad87] P. Wadler. Strictness analysis over non-flat domains. In *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[Wad90] P. Wadler. Linear types can change the world! In *Proceedings of the Working Conference on Programming Concepts and Methods*, pages 385–407. Israel, North-Holland, Amsterdam, 1990.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.