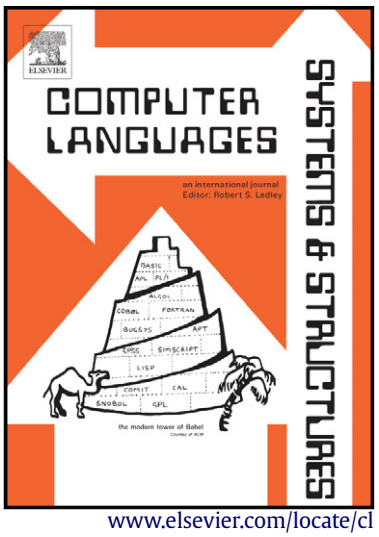

www.elsevier.com/locate/cl

Derivation and Inference of Higher-Order Strictness Types

Sjaak Smetsers, Marko van Eekelen

PII: S1477-8424(15)00051-2
DOI: http://dx.doi.org/10.1016/j.cl.2015.07.004
Reference: COMLAN182

To appear in: *Computer Languages, Systems & Structures*

Received date: 26 February 2015
Revised date: 2 June 2015
Accepted date: 30 July 2015

Cite this article as: Sjaak Smetsers, Marko van Eekelen, Derivation and Inference of Higher-Order Strictness Types, *Computer Languages, Systems & Structures,* http://dx.doi.org/10.1016/j.cl.2015.07.004

Derivation and Inference of Higher-Order Strictness Types

Sjaak Smetsers[a,*], Marko van Eekelen[a,b]

[a] *Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 212 6525 EC Nijmegen, The Netherlands*
[b] *Computer Science Department, Open University of the Netherlands, Valkenburgerweg 177 6419 AT Heerlen, The Netherlands*

Abstract

We extend an existing first-order typing system for strictness analysis to the fully higher-order case, covering both the derivation system and the inference algorithm. The resulting strictness typing system has expressive capabilities far beyond that of traditional strictness analysis systems. This extension is developed with the explicit aim of formally proving soundness of higher-order strictness typing with respect to a natural operational semantics. A key aspect of our approach is the introduction of a proof assistant at an early stage, namely during development of the proof. As such, the theorem prover aids the design of the language theoretic concepts. The new results in combination with their formal proof can be seen as a case study towards the achievement of the long term PoplMark Challenge. The proof framework developed for this case study can furthermore be used in other typing system case studies.

Keywords: strictness analysis, lambda calculus, typing, operational semantics, automated theorem proving.

1. Introduction

In this paper we present a type-theoretic approach to strictness analysis covering both type derivation and type inference[1]. The typing system includes higher-order types as well as user-defined recursive data types. One of our objectives is to formally prove that the higher-order strictness typing is sound with respect to a natural operational semantics. More specifically, we propose to use proof assistants not only for the re-construction of hand-written proofs, but moreover to introduce the tool *during the development* of language theoretic

*Corresponding author
Email addresses: S.Smetsers@cs.ru.nl (Sjaak Smetsers), M.vanEekelen@cs.ru.nl (Marko van Eekelen)

[1] This paper is an extension of an improved version of an earlier conference paper [SvE12] which covers type derivation only.

concepts. By introducing the tool in this stage, the consistency of technical concepts can be verified whilst designing them. This is accomplished by checking properties linking these concepts. This paper and the accompanying proof files comprise examples of such concepts and properties. Inaccuracies or mistakes made during the development were most often detected in an early stage, avoiding time consuming and inevitably failing attempts to construct a correctness proof of the main properties. This approach was used for the soundness proof of a non-standard typing system for a simple functional programming language. We combine standard Hindley-Milner typing with strictness information, specifying termination properties of higher-order functions. 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.

Combining standard typing with some form of input/ouput analysis is quite common. We mention a few examples. *Substructural type systems* ([Wal04]) regulate the order and number of uses of data by ensuring that some values be used at most once, at least once, or exactly once. E.g., linear typing systems (such as uniqueness typing) can be used to identify *unique objects*. These unique objects are suitable for *compile-time garbage collection* which is essential for incorporating destructive updates in functional languages (e.g., See [BS96, TWM95]). Security-typed languages ([VIS96], for instance) track information flow within programs to enforce security properties such as data confidentiality and integrity. This information can be used to prevent unintentional information leaks. [SvEvK09] describes a dedicated typing system for predicting the heap space usage of first-order, strict functional programs. This information can be used in several ways, most notably to ensure that a program allocates sufficient free memory.

Another view on this paper is that it reports on a case study of computer aided verification of theories about syntactic objects. Syntactic theories such as *operational semantics* and *type systems* play an important role in the (static) analysis of computer programs and the construction of reliable implementations of programming languages. The usability and reliability of syntactic techniques can undoubtedly be improved by using automated proof assistants. This need is recognized by many researchers. Most notably, the PoplMark Challenge [ABF$^+$05] calls for experiments on verifications of metatheory and semantics using proof tools. The concrete proposal is to formalize existing proofs of properties of type systems with different proof assistants. The long term goal is far more ambitious. It envisions "... a future in which the papers in conferences such as Principles of Programming Languages (POPL) and the International Conference on Functional Programming (ICFP) are routinely accompanied by mechanically checkable proofs of the theorems they claim."

The contribution of our work is threefold. The first contribution is the formalization of a non-standard typing system for strictness analysis of functional programs covering both the derivation system and the type inference algorithm. A first-order version of this typing system was presented in [BS07]. Compared to traditional strictness analyzers, it has two main advantages. Firstly, it can

be combined with ordinary typing: the compiler does not require an additional analysis phase. Secondly, it avoids fixed point computations, resulting in a much more efficient implementation. In this paper we augment first order typing with function types, in effect making it fully higher-order. Compared to common strictness analyses, the resulting system has an additional advantage: it permits the specification and derivation of strictness properties *between* the function arguments. We prove that our system is *sound* with respect to a given natural operational semantics. Thereafter, we discuss the extension of the system needed to deal with recursive data-types.

Secondly, it can be seen as a methodological experiment. We assess the usability of theorem provers for formalizing complex semantical issues, not only after the manual construction of the proofs, but especially during the development of basic theory. The complexity of the typing system in our case study is of a similar level as that of the PoplMark challenge. However, the main proof methods are not known in advance, as is the case in the challenges, but are to be developed during the proof process.

Finally, the PVS formalization can be used as a framework for developing other metatheoretical concepts. The framework can be used as a basis for developing other type based analyses together with their formal soundness proof, living up to the ambition of the long term PoplMark Challenge.

Overview

Section 2 introduces the core language used in this paper. Basic aspects of strictness are treated in Section 3 including semantic interpretations of (higher-order) strictness types. The derivation rules for deriving higher-order strictness types are given in Section 4. The soundness proof of this typing system is treated informally in Section 5. The formal proof is described in Section 6. Type inference is covered in Section 7. Recursive data structures are dealt with in Section 8. The paper concludes with a discussion of related work in Section 9 and with concluding remarks and ideas for future work in Section 10.

2. Extended Lambda Calculus

In this section we introduce the core functional language used throughout the paper. This language captures essential aspects such as basic values, abstraction, application, data constructors and destructors, and recursion.

2.1. Syntax

Definition 1. *Let* $\mathbb{V} = \{x, y, z, x_0, x_1, \ldots\}$ *be an infinite set of* term variables

- *The set* Λ *of* (lambda) expressions *is defined by the following abstract syntax.*

$$\begin{array}{rcl} \Lambda & ::= & \mathbb{V} \mid \Box \mid \lambda\mathbb{V}.\Lambda \mid \Lambda\,\Lambda \mid \langle \Lambda, \Lambda \rangle \mid \mathsf{fst}\,\Lambda \mid \mathsf{snd}\,\Lambda \mid \\ & & \mathsf{inl}\,\Lambda \mid \mathsf{inr}\,\Lambda \mid \mathsf{case}\,\Lambda\,\mathsf{of}\,\Lambda\,\mathsf{or}\,\Lambda \mid \mu\mathbb{V}.\Lambda. \end{array}$$

$$\frac{}{\Box \Downarrow \Box}(unit) \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}(abs)$$

$$\frac{M \Downarrow \lambda x.B \qquad B[x \leftarrow N] \Downarrow V}{M\,N \Downarrow V}(appl)$$

$$\frac{M \Downarrow \langle X, Y\rangle \quad X \Downarrow V}{\mathsf{fst}\,M \Downarrow V}(fst) \qquad \frac{M \Downarrow \langle X, Y\rangle \quad Y \Downarrow V}{\mathsf{snd}\,M \Downarrow V}(snd)$$

$$\frac{}{\langle X, Y\rangle \Downarrow \langle X, Y\rangle}(pair)$$

$$\frac{}{\mathsf{inl}\,L \Downarrow \mathsf{inl}\,L}(inl) \qquad \frac{}{\mathsf{inr}\,R \Downarrow \mathsf{inr}\,R}(inr)$$

$$\frac{S \Downarrow \mathsf{inl}\,L \qquad G\,L \Downarrow V}{\mathsf{case}\,S\,\mathsf{of}\,G\,\mathsf{or}\,H \Downarrow V}(case\text{-}L) \qquad \frac{S \Downarrow \mathsf{inr}\,R \qquad H\,R \Downarrow V}{\mathsf{case}\,S\,\mathsf{of}\,G\,\mathsf{or}\,H \Downarrow V}(case\text{-}R)$$

$$\frac{M[x \leftarrow \mu x.M] \Downarrow V}{\mu x.M \Downarrow V}(fix)$$

Figure 1: Evaluation rules

- *The set of free variables of M is denoted by* $\mathrm{FV}(M)$. *Let* $\vec{x} = (x_1, \ldots, x_n)$. *We write* $\Lambda^{\vec{x}}$ *for the set of* λ*-terms closed by* $\vec{x}$, *i.e.,* $\{M \in \Lambda | \mathrm{FV}(M) \subseteq \vec{x}\}$. *We write* Λ^{o} *instead of* $\Lambda^{()}$ *(expressions with no free variables, so called* closed expressions*).*

The constructor $\Box$ represents all basic values (integers, booleans, etc.). Pairs are constructed using the expression $\langle e_x, e_y\rangle$, and destructed using projections $\mathsf{fst}\,e$ and $\mathsf{snd}\,e$. The constructors **inl** and **inr** are sum left and right injections of the disjoint unions, whereas **case** is the destructor for these expressions.

In the sequel, we use capital letters like $M, N, X, Y, B, \ldots$ as meta-variables to range over lambda expressions.

2.2. Semantics

We will describe an evaluation in which computations are done by successive substitutions or replacements (*call-by-name*). With some adjustments to the syntax it would have been possible to incorporate proper sharing, with a *call-by-reference* semantics in the style of [Lau93]. Since we do not consider this extension to be essential for strictness analysis, we will describe a system without sharing.

The *value* V of a closed expression M is defined via a standard natural 'big step' operational semantics expressed as judgements of the form $M \Downarrow V$. This evaluation will yield an (also closed) expression in head-normal form.

Definition 2. *Let* $M \in \Lambda^{o}$.

- *We write* $M \Downarrow V$*, and say that* M evaluates to V *if this statement is derivable using the rules in Fig. 1.*
- M *is* defined *or* convergent *(notation* $M \Downarrow$*) if* $M \Downarrow V$ *for some value* V*. Otherwise* M *is* undefined *or* divergent *(notation* $M \Uparrow$*).*
- *The set of undefined (closed) expressions (i.e.,* $\{M \in \Lambda^o | M \Uparrow\}$*) is denoted by* $\mathbb{O}$*.*

Lemma 1. $M \Downarrow V$ *and* $M \Downarrow W \quad \Rightarrow \quad V = W$

$\mathbb{O}$ contains a canonical inhabitant $\mu x.x$, commonly denoted as $\bot$, that will be used to introduce *finite unfoldings*.

Definition 3. *Let* $F_x \in \Lambda^x$*. The* n^{th} *(finite) unfolding (notation* F_x^n*) is defined inductively by:*

$$F_x^0 = \bot \qquad F_x^{n+1} = F_x[x \leftarrow F_x^n]$$

The following property (the so-called *syntactical continuity property*, formally proved in [Sme10]) relates the evaluation of closed fix-expressions to the evaluation of finite unfoldings, and vise versa.

Proposition 1. *Let* $x, y \in \mathbb{V}$*, and* $C_y \in \Lambda^y$ *and* $F_x \in \Lambda^x$*.*

$$C_y[y \leftarrow \mu x.F_x] \Downarrow \quad \Leftrightarrow \quad \exists m \geq 0 : C_y[y \leftarrow F_x^m] \Downarrow$$

A disadvantage of a 'big step approach' is that reasoning about individual evaluation steps can be awkward. To circumvent this problem the following equivalence relation appears to be useful.

Definition 4. *Two expressions* $M, N \in \Lambda^o$ *are* reduction equivalent *(notation* $M =_\beta N$*) if for all* $H \in \Lambda^o$

$$M \Downarrow H \quad \Leftrightarrow \quad N \Downarrow H$$

3. Strictness

Plain strictness is usually defined as follows.

Definition 5. *Let* $\vec{x} = (x_1, \ldots, x_n)$*. An expression* $E \in \Lambda^{\vec{x}}$ *is* strict in x_i *(*$1 \leq i \leq n$*) if for all* $\vec{A} \in (\Lambda^o)^n$

$$A_i \Uparrow \quad \Rightarrow \quad E[\vec{x} \leftarrow \vec{A}] \Uparrow$$

A drawback of this notion of strictness is the lack of compositionality: strictness of a compound expression cannot always be determined by combining strictness of its constituents. For example the expression $\mathbf{fst}\, x$ is strict in x and the expression $\langle x, y \rangle$ not (and, of course, also not strict in y). However the compound expression $\mathbf{fst}\, \langle x, y \rangle$ *is* strict in x. Our aim is to refine this notion of

strictness in such a way that the evaluation properties of expressions are captured more accurately. For instance, the function **fst** not only evaluates its argument to head-normal form, but successively also evaluates the first component of the resulting pair. Moreover, the expression $\langle x, y\rangle$ is strict in x if it appears in a context that not only needs a pair but also the value of the first component of that pair (or of the second component to become strict in y instead of x), as is the case in our example. These *evaluation contexts* will be expressed as *strictness types*.

3.1. Strictness Types

A strictness type is a standard type annotated with so-called *strictness attributes*. The idea is to formulate strictness of E in x by a typing statement

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

The refinement mentioned is accomplished by admitting attributes to more than only the outermost level of a type. For example,

1. $x{:}(\sigma^! \times \tau)^! \vdash \mathbf{fst}\, x : \sigma^!$
2. $x{:}\sigma^! \vdash \langle x, x\rangle : (\sigma^! \times \sigma)^!$

Typing (1) expresses that **fst** will evaluate its first argument as indicated above. Typing (2) expresses that in a context in which the first component of a pair is needed (which is indicated by the result type $(\sigma^! \times \sigma)^!$) the expression itself becomes strict in x. Observe that if the second component of the pair was needed, a typing for (2) would be $x{:}\sigma^! \vdash \langle x, x\rangle : (\sigma \times \sigma^!)^!$. To avoid confusion, we now introduce an explicit notation for the absence of strictness information, namely ? (pronounced as *lazy*).

Definition 6. • *Let* $\Phi = \{\alpha, \beta, \alpha_0, \alpha_1, \ldots\}$ *be an infinite set of* type variables*, and* $A = \{!, ?\}$ *the set of* strictness attributes *(ranged over by meta-variables* u, v*),* $\mathbb{T} = \Sigma \cup \Pi$ *denotes the set* strictness types*. Here* Σ *and* Π *are defined by the following abstract syntax.*

$$\begin{array}{rcl} \Sigma & ::= & \Pi^A \\ \Pi & ::= & \Phi \mid \mathbb{1} \mid \Sigma \to \Sigma \mid \Sigma \times \Sigma \mid \Sigma + \Sigma \end{array}$$

The outermost attribute of $S \in \Sigma$ *is denoted by* $[S]$*. Moreover,* $\lfloor S \rfloor$ *denotes the type obtained from* S *by removing the outermost attribute. Hence,* $\lfloor S \rfloor^{[S]} = S$*.*

To avoid brackets, we will write $(\ldots \to \ldots)^u$ *as* $\ldots \xrightarrow{u} \ldots$*.*

• *Let* $|T|$ *denote the 'stripped' version of* T*, i.e.,* T *without any strictness attributes. We consider two types* T_1, T_2 *as* equivalent *(notation* $T_1 \sim T_2$*) if* $|T_1| \equiv |T_2|$*. So, types are equivalent if their underlying standard types are identical.*

Definition 7. • *Strictness attributes are ordered as follows:* $! \leq ?$

- *This ordering on attributes induces the following* coercion relation *on* $\mathbb{T}$.

$$\begin{array}{rcl} u \leq v \text{ and } \sigma \leq \tau & \Rightarrow & \sigma^u \leq \tau^v \\ & & \mathbb{1} \leq \mathbb{1} \\ & & \alpha \leq \alpha \\ S_1 \leq S_2 \text{ and } T_1 \leq T_2 & \Rightarrow & S_2 \to T_1 \leq S_1 \to T_2 \text{ and} \\ & & S_1 \times T_1 \leq S_2 \times T_2 \text{ and} \\ & & S_1 + T_1 \leq S_2 + T_2 \end{array}$$

  Note the contravariance in the first argument of $\to$.

- *The* infimum *of two attributes* u, v *(notation* $u \sqcap v$*) is the miminum of* u, v *w.r.t.* $\leq$

- *The predicate* inf *on* $(\mathbb{T}, \mathbb{T}, \mathbb{T})$ *is defined by induction:*

$$\begin{array}{rcl} \mathsf{inf}(\sigma^u, \tau^v, \rho^w) & = & u = v \sqcap w \text{ and } \mathsf{inf}(\sigma, \tau, \rho) \\ \mathsf{inf}(\alpha, \alpha, \alpha) & = & \mathsf{true} \\ \mathsf{inf}(\mathbb{1}, \mathbb{1}, \mathbb{1}) & = & \mathsf{true} \\ \mathsf{inf}(S \to T, S_1 \to T_1, S_2 \to T_2) & = & S = S_1 = S_2 \text{ and } T = T_1 = T_2 \\ \mathsf{inf}(S \times T, S_1 \times T_1, S_2 \times T_2) & = & \mathsf{inf}(S, S_1, S_2) \text{ and } \mathsf{inf}(T, T_1, T_2) \\ \mathsf{inf}(S + T, S_1 + T_1, S_2 + T_2) & = & \mathsf{inf}(S, S_1, S_2) \text{ and } \mathsf{inf}(T, T_1, T_2) \\ \mathsf{inf}(\cdot, \cdot, \cdot) & = & \mathsf{false} \end{array}$$

  The last rule should only be used if none of the othes rules applies, i.e. if t_1, t_2 *and* t_3 *are not equivalent.*

The inf predicate is used in our typing system to combine typing assumptions of different occurrences of the same expression variable, commonly called *contraction*. Consider, for example the following function:

$$\lambda x.+\,(\mathbf{fst}\,x)\,(\mathbf{snd}\,x)$$

If we assume that $+$ is strict in both arguments (say of type N), then the first occurrence of x will get type $(N^! \times N^?)^!$, and the second $(N^? \times N^!)^!$. The occurrences are combined by taking the infimum of their types, being $(N^! \times N^!)^!$.

3.2. Semantics of types

The meaning of a (strictness) type S is formalized by interpreting S as a subset of Λ^o. The 'standard' interpretation $[\![S]\!]$ contains all expressions that either evaluate to a value of type $|S|$ or diverge. For example $[\![(\mathbb{1}^u \times \mathbb{1}^v)^w]\!]$ (the standard interpretation does not depend on concrete attribute values) contains all expressions that are either in $\mathbb{0}$ or evaluate to $\langle \Box, \Box \rangle$, $\langle \Box, - \rangle$ or to $\langle -, \Box \rangle$. Here, we use $-$ to indicate that the corresponding expression diverges.

Definition 8 (Standard type interpretation). • *Let* $A, B \subseteq \Lambda^o$.

$$\begin{array}{rcl} \underline{\mathbb{1}} & = & \{M \in \Lambda^o \mid M \Downarrow \Box\} \\ A \underline{\times} B & = & \{M \in \Lambda^o \mid \exists a \in A, b \in B : M \Downarrow \langle a, b \rangle\} \\ A \underline{+} B & = & \{M \in \Lambda^o \mid \exists a \in A : M \Downarrow \mathbf{inl}\,a \text{ or } \exists b \in B : M \Downarrow \mathbf{inr}\,b\} \\ A \underline{\to} B & = & \{M \in \Lambda^o \mid \forall a \in A : (M\,a) \in B\} \end{array}$$

$$
\begin{array}{rcl}
[\![\sigma^u]\!] & = & \mathbb{O} \cup [\![\sigma]\!] \\
[\![\alpha]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & \underline{\mathbb{1}} \\
[\![S \to T]\!] & = & [\![S]\!] \mathbin{\underline{\to}} [\![T]\!] \\
[\![S \times T]\!] & = & [\![S]\!] \mathbin{\underline{\times}} [\![T]\!] \\
[\![S + T]\!] & = & [\![S]\!] \mathbin{\underline{+}} [\![T]\!]
\end{array}
$$

Figure 2: Standard semantics of types

- *The interpretation $[\![S]\!]$ are inductively defined in Fig. 2.*

For strictness types, we will need two more interpretations: $[\![\cdot]\!]_?$ and $[\![\cdot]\!]_!$. Let S be a strictness type S. Then $[\![S]\!]_?$ denotes *all* expressions that inhabit type $|S|$, including $\mathbb{O}$. $[\![S]\!]_!$ denotes the set of expressions that diverge when used in a context with type S. For instance, $[\![\mathbb{1}^!]\!]_?$ contains all expressions that either evaluate to $\Box$ or diverge. $[\![\mathbb{1}^?]\!]_?$ is the same set. $[\![\mathbb{1}^!]\!]_!$ is equal to $\mathbb{O}$, whereas $[\![\mathbb{1}^?]\!]_!$ is empty, since type $\mathbb{1}^?$ used in the latter corresponds to a lazy context. Slightly more complicated is the following example in which we take $(\mathbb{1}^! + \mathbb{1}^?)^!$ as S. Now $[\![S]\!]_?$ contains all expressions that either diverge or evaluate to $\mathbf{inl}\, I$ or to $\mathbf{inl}\, R$, with $I \in [\![\mathbb{1}^!]\!]_?, R \in [\![\mathbb{1}^?]\!]_?$. $[\![S]\!]_!$, however, contains besides all divergent expressions only expressions evaluating to $\mathbf{inl}\, I$, with $I \in [\![\mathbb{1}^!]\!]_! = \mathbb{O}$. The case $\mathbf{inl}\, R$ is impossible since this would require that $R \in [\![\mathbb{1}^?]\!]_! = \emptyset$.

Definition 9 (Strictness type interpretation). *The interpretations $[\![S]\!]_?$ and $[\![S]\!]_!$ are defined by mutual induction in Fig. 3.*

The interpretations $[\![\cdot]\!]$ and $[\![\cdot]\!]_?$ are almost identical. They only differ in the way function types are treated. The function $\lambda x.\Box$, for instance, is a member of $[\![\alpha^! \stackrel{!}{\to} \mathbb{1}^!]\!]$, but not a member of $[\![\alpha^! \stackrel{!}{\to} \mathbb{1}^!]\!]_?$. The latter is due to the additional requirement that any expression $E \in [\![\alpha^! \to \mathbb{1}^!]\!]_?$ should also be a member of $[\![\alpha^!]\!]_! \mathbin{\underline{\to}} [\![\mathbb{1}^!]\!]_!$, which is not the case for $\lambda x.\Box$: if we substitute $\bot$ ($\in [\![\alpha^!]\!]_!$) for x we get $\Box$ as a result which is not an element of $[\![\mathbb{1}^!]\!]_! = \mathbb{O}$. We will elaborate further on this issue in Section 5.

The following property (based on finite unfoldings, see Definition 3) provides an induction scheme for proofs in which fixed point expressions are involved; e.g., see Theorem 1.

Proposition 2. *Let $T \in \mathbb{T}$, and $x \in \mathbb{V}, F_x \in \Lambda^x$. Then, for all $s \in \{!, ?\}$*

$$(\forall n \in \mathbb{N} : F_x^n \in [\![T]\!]s) \quad \Rightarrow \quad \mu x.F_x \in [\![T]\!]s$$

$$
\begin{array}{rcl}
[\![\sigma^u]\!]_? & = & \mathbb{O} \cup [\![\sigma]\!], \text{ if } u =? \\
 & = & \mathbb{O} \cup [\![\sigma]\!]_?, \text{ if } u =! \\
[\![\alpha]\!]_? & = & \emptyset \\
[\![\mathbb{1}]\!]_? & = & \underline{\mathbb{1}} \\
[\![S \to T]\!]_? & = & [\![S]\!]_? \mathrel{\underline{\to}} [\![T]\!]_? \cap [\![S]\!]_! \mathrel{\underline{\to}} [\![T]\!]_! \\
[\![S \times T]\!]_? & = & [\![S]\!]_? \mathrel{\underline{\times}} [\![T]\!]_? \\
[\![S + T]\!]_? & = & [\![S]\!]_? \mathrel{\underline{+}} [\![T]\!]_?
\end{array}
$$

$$
\begin{array}{rcl}
[\![\sigma^u]\!]_! & = & \emptyset, \text{ if } u =? \\
 & = & \mathbb{O} \cup [\![\sigma]\!]_!, \text{ if } u =! \\
[\![\alpha]\!]_! & = & \emptyset \\
[\![\mathbb{1}]\!]_! & = & \emptyset \\
[\![S \to T]\!]_! & = & [\![S]\!]_? \mathrel{\underline{\to}} [\![T]\!]_! \\
[\![S \times T]\!]_! & = & [\![S]\!]_! \mathrel{\underline{\times}} [\![T]\!]_? \cup [\![S]\!]_? \mathrel{\underline{\times}} [\![T]\!]_! \\
[\![S + T]\!]_! & = & [\![S]\!]_! \mathrel{\underline{+}} [\![T]\!]_!
\end{array}
$$

Figure 3: Semantics of strictness types

The proof of this property in which Proposition 1 plays a crucial role, is quite complex. In PVS it necessitates approximately 1000 proof steps in addition to several non-trivial helper lemmas. The complexity is caused by the fact that our purely syntactical approach requires tedious manipulations of various constructs. In a formalization on paper this would have been barely feasible.

Both interpretations are closed under beta-equivalence.

Proposition 3. *Let $M =_\beta N$. Then, for all strictness types T, and $s \in \{!, ?\}$:*

$$M \in [\![T]\!]s \quad \Leftrightarrow \quad N \in [\![T]\!]s$$

4. Strictness typing

In this section we present a type system for deriving strictness information of terms and formally prove that this system is *sound*. Soundness here means that if a term M can be typed with strictness type S, then indeed M is a member of both $[\![S]\!]_?$ and $[\![S]\!]_!$. In essence, strictness typing can be characterized as a backwards analysis (E.g., see [DW90]): strictness properties are determined by relating the effect of demands on the arguments to the effect of demands on the result.

$$\frac{S \le T}{\Gamma^?, x{:}T \vdash x : S}(var) \qquad \frac{}{\Gamma^? \vdash \Box : \mathbb{1}^u}(unit)$$

$$\frac{\Gamma_1 \vdash M : S \overset{[R]}{\to} R \quad \Gamma_2 \vdash N : S \quad [R] \le [S] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash M\,N : R}(app)$$

$$\frac{\Gamma, x{:}S \vdash B : R \quad u \le [R]}{\Gamma \vdash \lambda x.B : S \overset{u}{\to} R}(abs)$$

$$\frac{\Gamma_1 \vdash X : S \quad \Gamma_2 \vdash Y : T \quad u \le [S] \sqcap [T] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \langle X, Y \rangle : (S \times T)^u}(pair)$$

$$\frac{\Gamma \vdash P : (S \times T)^{[S]} \quad [T] =?}{\Gamma \vdash \mathbf{fst}\,P : S}(fst) \qquad \frac{\Gamma \vdash P : (S \times T)^{[T]} \quad [S] =?}{\Gamma \vdash \mathbf{snd}\,P : T}(snd)$$

$$\frac{\Gamma \vdash L : S \quad u \le [S]}{\Gamma \vdash \mathbf{inl}\,L : (S+T)^u}(inl) \qquad \frac{\Gamma \vdash R : T \quad u \le [T]}{\Gamma \vdash \mathbf{inr}\,R : (S+T)^u}(inr)$$

$$\frac{\Gamma_1 \vdash I : (S+T)^{[U]} \qquad \begin{array}{c}\Gamma_2 \vdash L : S \overset{[U]}{\to} U \\ \Gamma_2 \vdash R : T \overset{[U]}{\to} U\end{array} \qquad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \mathbf{case}\,I\,\mathbf{of}\,L\,\mathbf{or}\,R : U}(case)$$

$$\frac{\Gamma, x{:}T \vdash B : S \quad S \le T}{\Gamma \vdash \mu x.B : S}(fix)$$

Figure 4: Rules for strictness type assignment

Definition 10.
- *A* basis *(or* environment*) is a finite set of declarations of the form $x : S$, where $x \in \mathbb{V}, S \in \mathbb{T}$. For a given basis, all variables are assumed to be distinct. We will sometimes use the 'functional notation' $\Gamma(x)$ to obtain the type assigned to x by Γ.*
- *By $\Gamma^?$ we denote a* lazy basis *containing only declarations of the form $x : \sigma^?$.*
- *Two bases Γ_1, Γ_2 are* equivalent*, denoted as $\Gamma_1 \sim \Gamma_2$, if for each x one has $\Gamma_1(x) \sim \Gamma_2(x)$.*
- *The* inf *predicate for types extends to (equivalent) bases in a straightforward manner: Let $\Gamma \sim \Gamma_1 \sim \Gamma_2$. Then $\mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)$ if $\mathsf{inf}(\Gamma(x), \Gamma_1(x), \Gamma_2(x))$ for all x.*

Definition 11. *A* strictness typing statement *is an expression of the form $\Gamma \vdash M : S$, where Γ is a basis. Such a statement is* valid *if it can be derived by using the rules in Fig. 4.*

We briefly focus on these derivation rules. One should recall that in each of these rules strictness can only appear if the context type, as given by the

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha^! \leq \alpha^!}{x{:}\alpha^!, y{:}\beta^? \vdash x : \alpha^!}(var) \qquad ! \leq [\alpha^!]}{x{:}\alpha^! \vdash \lambda y.x : \beta^? \xrightarrow{!} \alpha^! \qquad ! \leq [\beta^? \xrightarrow{!} \alpha^!]}(abs)}{\vdash \lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!}(abs)}$$

Figure 5: A derivation for $\vdash \lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!$

conclusion, is strict itself. As such, one could say that 'strictness propagates outwards'.

The rule for variables (*var*) enforces that each strictness assumption $x{:}\sigma^!$ in the environment should be 'consumed' by a strict occurrence of x (otherwise, the premise of this rule cannot be valid). Since unit (*unit*) represents the result of a computation (an expression in head normal form), the complete basis has to be lazy. In the rule for application (*app*), the function expression is always evaluated (provided that the application itself appears in a strict context), explaining the $[R]$ attribute on the arrow. The inequality $[R] \leq [S]$ ensures that the argument can only be strict if the context is strict. In essence, this abstraction rule (*abs*) reflects the characterisation of strictness as indicated at the beginning of Sec. 3.1 The data constructor rules (*pair*, *inl*, *inr*) all involve expressions in head normal form. Strictness properties for expressions appearing under these constructors are directly adopted from their corresponding context type. The typing rules for projections (*fst*, *snd*) have already been discussed at the beginning of the previous section. The case-rule (*case*) is more subtle. Recall that the evaluation of an expression **case** I **of** L **or** R results in the evaluation of I first (explaining the attribute $[U]$ on the corresponding sum type), followed by the evaluation of either L (applied to the left component of the result of I) or R (applied to the right component), but not both. So **case** I **of** L **or** R can only be strict in a variable x if either I is strict in x or *both* L and R are strict in x. The latter is accomplished by supplying the type derivations L and R with the same basis Γ_2. In the case of a recursive definition (*fix*), say of x, all occurrences of x are not necessarily strict. Indeed, x itself will often appear in a lazy context. This explains the inequality $S \leq T$.

To prepare for type inference, the type assignment system is formulated in a fully syntax directed fashion. As such, non-structural rules, such as contraction, subsumption and weakening (that are usually defined separately), have been incorporated in the structural rules of the system.

As an example, a strictness type derivation for the expression $\lambda x.\lambda y.x$ is given in Fig. 5. More examples can be found in Section 6.

5. Soundness

In this section we demonstrate that our typing system is sound. As previously stated, plain strictness is formulated solely in terms of undefinedness of

a function argument as a consequence of undefinedness of the function result. However in our system strictness properties of function arguments can influence each other. Consider, for example, the function $\mathsf{AP} = \lambda f.\lambda x.f\,x$. Then strictness of AP in x depends on the strictness properties of the other argument f. If AP is applied to a strict function then the result will be strict in x. This is expressed by the following valid strictness typing for AP[2]

$$\mathsf{AP} :: (\alpha^! \to \beta^!) \to \alpha^! \to \beta^!$$

At this point it is important to see that Definition 5 is no longer adequate: whether $\mathsf{AP}\,F \perp \Uparrow$ also depends on F. More specifically, F should be a strict function, and not just any arbitrary expression as in Definition 5. E.g., if we take $\lambda x.\Box$ for F then $\mathsf{AP}\,F \perp \Downarrow \Box$.

Definition 12.

- *An* expression environment *is a function ρ from $\mathbb{V}$ to Λ^o. Such an environment can be lifted to Λ in the obvious way. The result of applying ρ to an expression E is denoted by E^ρ.*
- *Let Γ be a basis. An environment ρ* is valid for Γ *(notation $\rho \Vdash \Gamma$), if $\forall (x{:}S) \in \Gamma : \rho(x) \in [\![S]\!]_?$.*
- *Similarly, ρ* satisfies Γ *(notation $\rho \Vdash\!\!\!\Vdash \Gamma$) if $\exists (x{:}S) \in \Gamma : \rho(x) \in [\![S]\!]_!$.*

Now the soundness of the type system (with respect to the semantics given in Definition 9) can be formulated as follows:

Theorem 1.

$$\Gamma \vdash E : S \quad \Rightarrow \quad \forall \rho : \rho \Vdash \Gamma \quad \Rightarrow \begin{cases} E^\rho \in [\![S]\!]_? & (1) \\ \rho \Vdash\!\!\!\Vdash \Gamma \quad \Rightarrow \quad E^\rho \in [\![S]\!]_! & (2) \end{cases}$$

Observe that conclusion (2) is essentially a reformulation of Definition 5. The problem about the strictness dependencies between arguments is solved by allowing only *valid* environments. Take, for instance, the expression fx being the body of the AP function in the above example. A possible strictness typing (in which we have substituted $\mathbb{1}$ for both α and β) for this expression is:

$$f{:}\mathbb{1}^! \xrightarrow{!} \mathbb{1}^!, x{:}\mathbb{1}^! \vdash f\,x : \mathbb{1}^!$$

Soundness of this typing requires that any environment ρ for fx should be both valid for and satisfying $\Gamma = f{:}\mathbb{1}^! \xrightarrow{!} \mathbb{1}^!, x{:}\mathbb{1}^!$. If such a ρ substitutes $\perp$ for x it indeed satisfies Γ, since $\perp \in [\![\mathbb{1}^!]\!]_!$. However, the validity of ρ prohibits $\lambda x.\Box$ to be substituted for f, because $\lambda x.\Box$ is not a member of $[\![\mathbb{1}^! \xrightarrow{!} \mathbb{1}^!]\!]_?$, since $\lambda x.\Box \notin [\![\mathbb{1}^!]\!]_! \underline{\to} [\![\mathbb{1}^!]\!]_!$; see Definition 9.

[2] For clarity, we have omitted the strictness attributes on the arrows. A full typing can be found in Lemma 2.

6. The PVS Formalization

In this section we discuss the formalization of the strictness typing system and its soundness proof in PVS.

We will not assume any preliminary knowledge about PVS and, as in the previous sections, continue to use tool independent notations. The actual formalization is straightforward and barely uses PVS specific constructs. Therefore, it should be relatively easy to convert our PVS specification to other proof assistants like Coq or Isabelle.

Firstly, it must be determined how to represent the variable bindings occurring in abstraction and fixed point expressions. We have chosen the De Bruijn notation mainly because our previous work uses the same representation and the system in this paper does not require significant fine-tuning. In the De Bruijn notation variables are identified by *indices*: natural numbers indicating the number of abstractions which must be skipped in order to localize the corresponding binder. If the variable number exceeds the number of surrounding abstractions, the variable is considered free.

Definition 13. • *The set* Λ_B *of* lambda terms with the De Bruijn indices *is defined by the following abstract syntax.*

$$\begin{array}{rcl} \Lambda_B & ::= & \mathbb{N} \mid \Box \mid \lambda\Lambda_B \mid \Lambda_B\,\Lambda_B \mid \langle \Lambda_B, \Lambda_B \rangle \mid \mathsf{fst}\,\Lambda_B \mid \mathsf{snd}\,\Lambda_B \mid \\ & & \mathsf{inl}\,\Lambda_B \mid \mathsf{inr}\,\Lambda_B \mid \mathsf{case}\,\Lambda_B\,\mathsf{of}\,\Lambda_B\,\mathsf{or}\,\Lambda_B \mid \mu\Lambda_B \end{array}$$

- *The predicate* $closed_n(M)$ *checks whether none of the free variables of* M *exceeds* n*. The definition of this predicate is obvious.*

With the De Bruijn notation one can avoid alpha-conversion during evaluation. However, the substitution itself is more complicated because one has to prevent that in $M[x \leftarrow N]$ the free variables of a N get captured by the binders of M. This mandates an adjustment of the free variables of M. Usually the correction of N is performed by an auxiliary function, whereas M is adjusted on-the-fly (e.g., see [Kam01] for a formal definition of these operations).

The soundness property is formulated almost in exactly the same way as Theorem 1. We have proved (1) and (2) in this theorem simultaneously by induction on the derivation of $\Gamma \vdash E : S$. The different cases require on average 200 proof steps each, which sums up to approximately 2200 steps all together[3].

One of the advantages of having a full formalization is that one can actually prove that examples are indeed correct explaining why we have formulated them as a lemma.

Lemma 2. 1. $\lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!$

2. $\lambda f.\lambda x.f\ x : (\alpha^! \xrightarrow{!} \beta^!) \xrightarrow{!} \alpha^! \xrightarrow{!} \beta^!$

[3]The complete proof files can be downloaded from `www.cs.ru.nl/~sjakie/papers/strictnesstyping/`.

3. $\mu f.\lambda n.\mathsf{case}\, n\, \mathsf{of}\, \lambda x.\mathsf{inl}\, \square\, \mathsf{or}\, \lambda x.f\, (\mathsf{inr}\, x) : N^! \xrightarrow{!} N^!$
4. $\mu f.\lambda x.\lambda y.\mathsf{case}\, x\, \mathsf{of}\, \lambda z.f\, (\mathsf{inr}\, \square)\, y\, \mathsf{or}\, \lambda z.f\, y\, (\mathsf{inr}\, \square) : N^! \xrightarrow{!} N^! \xrightarrow{!} N^!$

The proof of these typing statements can also be found in the PVS files. In Example 3 and 4, N stands for the type $\mathbb{1}+\mathbb{1}$ which we use to represent natural numbers. The actual values of these numbers are not relevant: It suffices if we can distinguish 0 (represented as $\mathsf{inl}\,\square$) from all other numbers (represented as $\mathsf{inr}\,\square$). Example 3 resembles the factorial function (without the usual multiplication and subtraction), and Example 4 (showing a recursive function that is strict in both arguments) is taken from [CDG02].

6.1. Conducting the proof

Automated theorem proving is very time consuming. We estimate that the construction of the entire proof (including the development of the necessary theoretic concepts) comprises about six man-months work, maybe even more. On the other hand, it revealed mistakes which had been made in the previous (first-order) version of the type system. Moreover, extending the language with higher-order constructs made the system significantly more complex. For example, the treatment of function types in Def. 7 showed to be non-intuitive. Various attempts were made preceding the final version. The PVS formalization helped us by enabling quick verification of modifications in definitions.

Our work can be considered as a contribution in the spirit of [ABF$^+$05]. One of the PoplMark challenges is the treatment of variable binding. Most of the solutions that have been reported to this challenge are based on de Bruijn indices. Though [ABF$^+$05] argues that this representation introduces too much overhead in formal proofs, and should therefore be avoided, this is not confirmed by our approach or by any of these solutions. The low-level variable representation does not lead to any significant increase in the complexity of the proofs. With a few simple, and easy–to–prove auxiliary lemmas, one can effectively hide all implementation details. The main proof itself is not affected. In fact, the implicit bindings of De Bruijn's representation allow environments to be represented as simple lists rather than lists of pairs.

We believe that, besides obtaining full confidence, one of the main advantages of possessing a formal proof is the possibility to replay the proof in a tool. For instance, if one cannot immediately follow the given explanations, in principle one can fall back on the fully elaborated formal version.

7. Inferring Higher Order Strictness Types

In this section we will describe how strictness variants of traditional typings can be inferred effectively.

7.1. Standard type inference

The type derivation algorithm is based on the idea to split type reconstruction into two phases ([Wan87]): generation of requirements (in the form of *type*

equations) and fulfilling these requirements (in the form of a solution for the equations, i.e., a suitable substitution for the type variables). The solution is computed by means of a procedure called *unification*; see [Rob65].

Since the system is *fully syntax directed* (each syntactic construction has exactly one typing rule), the equations $\sigma \simeq \tau$ for an expression E can be generated by stepwise reconstruction of a type derivation for E, essentially leading to a type reconstruction for each subexpression of E.

We will illustrate this idea with an example. Consider the following expression.

$$\lambda p.\mathsf{case}\, p\, \mathsf{of}\, \lambda x.x\, \mathsf{or}\, \lambda y.\Box$$

For the outermost abstraction, two auxiliary variables variables are introduced, say α, β. The variable α is assigned to p whereas β becomes the result type of the case expression. The latter consists of three components. In order to reconstruct the type, two more variables are needed, say γ, δ. Typing the pattern leads to the equation $\alpha \simeq \gamma + \delta$. Both branches of the case consist of a lambda expression. Each lambda expression will again require new auxiliary variables. To simplify our example slightly, we assume that $\lambda x.x$ if typed with $\zeta \to \zeta$ and $\lambda y.\Box$ with $\eta \to \mathbb{1}$, both ζ, η fresh. Typing these branches will lead to the equations $\gamma \to \beta \simeq \zeta \to \zeta$ and $\delta \to \beta \simeq \eta \to \mathbb{1}$. Solving these three equations by unification gives the substitution

$$* = \{\alpha \leftarrow \mathbb{1} + \eta, \delta \leftarrow \eta, \zeta \leftarrow \mathbb{1}, \beta \leftarrow \mathbb{1}, \gamma \leftarrow \mathbb{1}\}$$

Applying this substitution gives the typing statement

$$\lambda p.\mathsf{case}\, p\, \mathsf{of}\, \lambda x.x\, \mathsf{or}\, \lambda y.\Box : \mathbb{1} + \eta \to \mathbb{1}$$

One can use this algorithm to show that type assignment has the *principal typing property*: if an expression E is typable, there exist a principal typing for E, i.e., a 'schematic' type of which all other typings can be obtained via instantiation of type variables.

The above typing for E is principal.

Principal Typing Theorem 1. *Principal typings can be computed effectively.*

Proof. See [BS96]. □

7.2. Strictness polymorphism

For strictness types we follow roughly the same procedure. Since strictness typing involves subtyping, we will generate both type equations and type *inequalities*.

Given an expression and a standard type, there can be several strictness variants of that typing. Therefore it is natural to consider strictness *schemes*. We use attribute variables $(a, b, \ldots)$ to denote schematic strictness types, in the same way as we use type variables to indicate schematic standard types. A concrete strictness typing consists of a proper instantiation of the attribute variables.

We will need to state dependencies between strictness attributes. We will express these as (finite) sets of attribute inequalities called *attribute environments*. The following example illustrates the use of these environments. The possible types of the lambda expression $\lambda x.\mathbf{fst}\, x$ can be expressed schematically as

$$(\alpha^a \times \beta^?)^c \xrightarrow{d} \alpha^b | \quad b \leq a, b \leq c, d \leq b$$

The attribute environment denotes restrictions on instantiations. E.g., $a :=!, b :=!, c :=!, d :=!$ and $a :=?, b :=!, c :=!, d :=!$ are valid instantiations, but, for example, $a :=?, b :=?, c :=!, d :=!$ is not, since $? \not\leq !$.

It is possible to add these 'polymorphic strictness types' and attribute environments to the formal typing system. A similar extension has been done in the uniqueness type system, see [BS96]. We will refrain from such an extension here, since it produces administrative overhead and does not contribute to the understanding of the typing algorithm. Instead, we will regard our strictness schemes as an abbreviation for the respective concrete strictness types obtained by instantiation.

During strictness type reconstruction we sometimes need to denote the 'minimum' of two strictness types or strictness attributes. This operation occurs whenever the typing rule contains an infimum of bases. In case this minimum is not directly computable (when attributes are schematic, or types are not yet equivalent) we continue the computation with a schematic minimum $S \sqcap T$. Eventually, this leads to schematic attributes of the form $a_1 \sqcap \ldots \sqcap a_k$. For example, $\lambda x.\langle x, x\rangle$ can be typed with

$$\lambda x.\langle x, x\rangle : \alpha^{a \sqcap b} \xrightarrow{f} (\alpha^c \times \alpha^d)^e | \quad f \leq e, e \leq c, e \leq d, c \leq a, d \leq b.$$

The attribute $a \sqcap b$ on the argument type together with the inequalities $c \leq a, d \leq b$ indicate that this argument can only become ! if either c or d is taken !.

7.3. Generating requirements

The requirements generated during type reconstruction consist of strictness type equations but also of strictness type inequalities (due to subtyping constraints $S \leq T$ in the rules for *var* and *fix*) and strictness attribute inequalities (due to attribute constraints $u \leq v$).

We can describe the generation of requirements as a function $\mathcal{S}$. It takes an expression E and a goal type T as input and produces a pair $\langle \Gamma, \mathcal{R}\rangle$ consisting of a basis Γ and a collection of inequalities $\mathcal{R}$. The latter collection is a triple consisting of a set of type equations, a set of type inequalities, and a set of attribute inequalities.

Definition 14. *The* strictness requirements generation function $\mathcal{S}$ *is defined inductively in Fig. 6. Union of results is to be taken componentwise.*

Some remarks: If the variable x in the rules for *fix* and *abstraction* does not occur in E there will be no declaration of x in the resulting basis. Hence, we cannot write the basis as $\Gamma \cup \{x{:}X\}$. In the case of *abstraction*, the type X is

$$
\begin{array}{rcl}
\mathcal{S}(x,T) & = & \langle\{x{:}\alpha^a\},\langle -,\{T\le\alpha^a\},-\rangle\rangle,\\
\mathcal{S}(\Box,T) & = & \langle\emptyset,\langle\{\lfloor T\rfloor\simeq\mathbb{1}\},-,-\rangle\rangle\\
\mathcal{S}(E\,E',T) & = & \langle\Gamma\sqcap\Gamma',\mathcal{R}\cup\mathcal{R}'\cup\\
 & & \quad\langle -,-,\{[T]\le a\}\rangle\rangle\\
 & & \text{where}\quad \mathcal{S}(E,\alpha^a\overset{[T]}{\to}T)=\langle\Gamma,\mathcal{R}\rangle\\
 & & \qquad\quad \mathcal{S}(E',\alpha^a)=\langle\Gamma',\mathcal{R}'\rangle\\
\mathcal{S}(\lambda x.E,T) & = & \langle\Gamma,\mathcal{R}\cup\langle\{\lfloor T\rfloor\simeq X\to\alpha^a\},-,\{[T]\le a\}\rangle\rangle\\
 & & \text{where } \mathcal{S}(E,\alpha^a)=\langle\Gamma\cup\{x{:}X\},\mathcal{R}\rangle,\\
\mathcal{S}(\langle E,E'\rangle,T) & = & \langle\Gamma\sqcap\Gamma',\mathcal{R}\cup\mathcal{R}'\cup\\
 & & \quad\langle\{\lfloor T\rfloor\simeq\alpha^a\times\beta^b\},-,\{[T]\le a\sqcap b\}\rangle\rangle\\
 & & \text{where}\quad \mathcal{S}(E,\alpha^a)=\langle\Gamma,\mathcal{R}\rangle\\
 & & \qquad\quad \mathcal{S}(E',\beta^b)=\langle\Gamma',\mathcal{R}'\rangle\\
\mathcal{S}(\mathsf{fst}\,E,T) & = & \mathcal{S}(E,(T\times\alpha^?)^{[T]})\\
\mathcal{S}(\mathsf{snd}\,E,T) & = & \mathcal{S}(E,(\alpha^?\times T)^{[T]})\\
\mathcal{S}(\mathsf{inl}\,E,T) & = & \langle\Gamma,\mathcal{R}\cup\langle\{\lfloor T\rfloor\simeq\alpha^a+\beta^b\},-,\{[T]\le a\}\rangle\rangle\\
 & & \text{where } \mathcal{S}(E,\alpha^a)=\langle\Gamma,\mathcal{R}\rangle,\\
\mathcal{S}(\mathsf{inr}\,E,T) & = & \langle\Gamma,\mathcal{R}\cup\langle\{\lfloor T\rfloor\simeq\alpha^a+\beta^b\},-,\{[T]\le b\}\rangle\rangle\\
 & & \text{where } \mathcal{S}(E,\beta^b)=\langle\Gamma,\mathcal{R}\rangle,\\
\mathcal{S}(\mathsf{case}\,I\,\mathsf{of}\,L\,\mathsf{or}\,R,T) & = & \langle\Gamma\sqcap\Gamma',\mathcal{R}\cup\mathcal{R}'\cup\mathcal{R}''\cup\\
 & & \quad\langle\Gamma'\simeq\Gamma'',-,-\rangle\rangle\\
 & & \text{where}\quad \mathcal{S}(I,(\alpha^a+\beta^b)^{[T]})=\langle\Gamma,\mathcal{R}\rangle,\\
 & & \qquad\quad \mathcal{S}(L,\alpha^a\overset{[T]}{\to}T)=\langle\Gamma',\mathcal{R}'\rangle,\\
 & & \qquad\quad \mathcal{S}(R,\beta^b\overset{[T]}{\to}T)=\langle\Gamma'',\mathcal{R}''\rangle,\\
\mathcal{S}(\mu x.E,T) & = & \langle\Gamma,\mathcal{R}\cup\langle -,\{T\le X\},-\rangle\rangle\\
 & & \text{where } \mathcal{S}(E,T)=\langle\Gamma\cup\{x{:}X\},\mathcal{R}\rangle
\end{array}
$$

Figure 6: Strictness requirements generation

taken to be $\alpha^?$, with α fresh. For *fix* expressions, the inequality $T \leq X$ can be omitted. The infimum of $\Gamma \sqcap \Gamma'$ consists of declarations $x{:}S \sqcap T$ for each variable x appearing in both Γ and Γ'. For a variable x that appears only in Γ (say with type S) and not in Γ', the infimum $\Gamma \sqcap \Gamma'$ contains just $x{:}S$. The case that x appears in Γ' and not in Γ is handled similarly. Analogously, if a variable x is not present in both Γ and Γ', the set of type equations $\Gamma \simeq \Gamma'$ will not contain an equation corresponding to x. Instead, $? \leq [S]$ should be added to the set attribute inequalities (where S is the type of x in either Γ or Γ').

7.4. The typing algorithm

The next step is to solve the inequalities obtained by $\mathcal{S}$. Our goal is to compute a 'principal strictness typing', i.e. a valid typing (possibly containing an attribute environment) of which all other concrete strictness typings can be obtained via instantiation.

The underlying 'standard part' of the requirements can easily be solved: The output of $\mathcal{S}$ can be converted into a set of standard type equations $\sigma \simeq \tau$, by ignoring all strictness attributes and by considering type inequalities and type infima as equations. Let E be an expression. Suppose we apply $\mathcal{S}$ to $\langle E, \alpha^a \rangle$, with α and a fresh. It is not difficult to show that the most general solution for the resulting collection of equations (after applying the above procedure) leads to a principal standard typing of E.

Principal Strictness Typing Theorem 1. *Principal strictness typings can be computed effectively.*

Proof (Sketch of the algorithm). Let E be an expression.

Step 1. Compute $\langle \Gamma, \mathcal{R} \rangle = \mathcal{S}(E, \alpha^a)$.

Step 2. Determine (by unification) a solution $*_0$ for the standard type equations that result from $\Gamma, \mathcal{R}$.

Step 3. Lift $*_0$ to a strictness type substitution $*$, i.e., for each $\alpha = \sigma$, the type σ is converted into a pseudo strictness type by decorating all subtypes of σ with fresh attribute variables.

Step 4. Extend the set of type attribute inequalities of $\mathcal{R}$ with the inequalities that arise from the type equations and inequalities in $\mathcal{R}$ after performing $*$. Let Δ be the result of this step.

Step 5. Now $\langle \Gamma^*, (\alpha^a)^*, \Delta \rangle$ is a principal typing for E. □

To obtain more legible types, one could simplify the result of step 4 by determining the restriction of Δ to attributes appearing in Γ^* and $(\alpha^a)^*$. We will not explain this procedure in more depth.

The principal strictness type can be seen as the 'best strictness type' (meaning 'as strict as possible') which is obtained by choosing ! for all attribute variables.

7.5. Example

In this section we will illustrate strictness type inference with a fully elaborated example. In the course of the procedure we will need to introduce fresh type and strictness variables. We will use numbers for type variables instead of Greek letters, and Roman letters for strictness variables. The application of $\mathcal{S}$ to an expression E and a fresh goal type 1^a will produce a basis and set of requirements. During this generation phase, a basis can grow, shrink or present declarations might change. The set of requirements, however, is only extended. To enhance readability, we do not explicitly collect all the generated requirements at each step. Instead, we only mention the newly created ones.

Consider the expression $E = \lambda x.\langle x, x\rangle$. We will show that the strictness type for E that was given before is indeed computed by our type inference algorithm.

Step 1. Compute $\langle \Gamma, \mathcal{R}\rangle = \mathcal{S}(E, 1^a)$

$$
\begin{array}{l}
\Rightarrow\ \mathcal{S}(\lambda x.\langle x, x\rangle, 1^a) \\
\quad \Rightarrow\ \mathcal{S}(\langle x, x\rangle, 2^b) \\
\quad\quad \Rightarrow\ \mathcal{S}(x, 3^c) \\
\quad\quad \Leftarrow\ \langle x{:}5^e, \langle -, \{3^c \le 5^e\}, -\rangle\rangle \\
\quad\quad \Rightarrow\ \mathcal{S}(x, 4^d) \\
\quad\quad \Leftarrow\ \langle x{:}6^f, \langle -, \{4^d \le 6^f\}, -\rangle\rangle \\
\quad \Leftarrow\ \langle x{:}5^e \sqcap 6^f, \langle \{2 \simeq 3^c \times 4^d\}, -, \{b \le c \sqcap d\}\rangle\rangle \\
\Leftarrow\ \langle -, \langle \{1 \simeq 5^e \sqcap 6^f \to 2^b\}, -, \{a \le b\}\rangle\rangle
\end{array}
$$

Step 2. Determine a substitution that solves the set of standard type equations that result from $\Gamma, \mathcal{R}$. In this example this step is almost trivial: a possible solution, say $*_0$, is:

$$*_0 = \{3 \leftarrow 4, 3 \leftarrow 5, 3 \leftarrow 6, 2 \leftarrow 3 \times 3, 1 \leftarrow 3 \to 3 \times 3\}$$

Step 3. Lifting $*_0$ to a strictness substitution $*$ normally requires new fresh strictness variables. However, in this example we can optimize this step by using the generated equations for variable 1 as well as for 2.

$$* = \{3 \leftarrow 4, 3 \leftarrow 5, 3 \leftarrow 6, 2 \leftarrow 3^c \times 3^d, 1 \leftarrow 3^{e \sqcap f} \to (3^c \times 3^d)^b\}$$

Step 4. Now the type inequalities are converted into attribute inequalities, and added to the attribute inequalities already present in $\mathcal{R}$. The resulting collection Δ of inequalities is:

$$\Delta = \{c \le e, d \le f, b \le c \sqcap d, a \le b\}$$

Step 5. Finally, we end up with the following typing for E. Note that we can write $b \le c \sqcap d$ as $b \le c, b \le d$

$$\lambda x.\langle x, x\rangle : 3^{e \sqcap f} \xrightarrow{a} (3^c \times 3^d)^b \mid \quad c \le e, d \le f, b \le c, b \le d, a \le b$$

8. 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 extension can serve as a bases for the treatment of other recursive data types.

Lists are built up from constructors **nil** (the empty list) and **cons**. We incorporate these constructs in our syntax together with a destructor called **list** leading to the following extension of Definition 1

$$\Lambda \quad ::= \quad \cdots \mid \mathsf{nil} \mid \mathsf{cons}\,\Lambda\,\Lambda \mid \mathsf{list}\,\Lambda\,\Lambda\,\Lambda$$

By $\mathbf{L}(\sigma)$ we denote the (standard) type of lists of σ objects.

The evaluation rules for these construct are straightforward: hnf-evaluation of list objects stops at the outermost constructor.

$$\frac{}{\mathsf{nil} \Downarrow \mathsf{nil}}(nil) \qquad \frac{}{\mathsf{cons}\,H\,T \Downarrow \mathsf{cons}\,H\,T}(cons)$$

$$\frac{L \Downarrow \mathsf{nil} \qquad N \Downarrow V}{\mathsf{list}\,L\,N\,C \Downarrow V}(list\text{-}N) \qquad \frac{L \Downarrow \mathsf{cons}\,H\,T \qquad C\,H\,T \Downarrow V}{\mathsf{list}\,L\,N\,C \Downarrow V}(list\text{-}C)$$

Besides plain hnf-evaluation, it is useful to distinguish other evaluation forms. The most common ones are *spine evaluation* and *full evaluation*. For instance, the function *length* computing the size of the list will enforce the complete evaluation of the list structure, but leave the elements unaffected. A function *sum* that sums all the elements of a list will not only evaluate the spine completely, but also all of its elements.

The evaluation contexts induced by these functions are again encoded in an appropriate strictness type. This leads to the following extension of Definition 6:

$$\Pi ::= \cdots \mid \mathbf{L}^{A}(\Sigma)$$

Only part of the types that can be constructed according to this syntax is well-formed. This is due to the fact that for data structures, strictness 'propagates outwards': in order to evaluate inner components of a data structure, the structure itself has to be evaluated before these components can be accessed.

For example, for a list of $\mathbb{1}$ elements we have 4 different valid strictness variants: $(\mathbf{L}^{?}(\mathbb{1}^{?}))^{?}$, $(\mathbf{L}^{?}(\mathbb{1}^{?}))^{!}$, $(\mathbf{L}^{!}(\mathbb{1}^{?}))^{!}$, and $(\mathbf{L}^{!}(\mathbb{1}^{!}))^{!}$, corresponding to no, hnf, spine and full evaluation, respectively.

The typing rules are extended according to our intended semantics.

$$\frac{}{\Gamma^{?} \vdash \mathsf{nil} : \mathbb{1}^{u}}(nil)$$

$$\frac{\Gamma_1 \vdash H : S \quad \Gamma_2 \vdash L : (\mathbf{L}^{v}(S))^{v} \quad u \leq v \quad v \leq [S] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \mathsf{cons}\,H\,L : (\mathbf{L}^{v}(S))^{u}}(cons)$$

$$\frac{\Gamma_1 \vdash L : (\mathbf{L}^{v}(S))^{[T]} \qquad \begin{array}{c}\Gamma_2 \vdash N : T \\ \Gamma_2 \vdash C : S \xrightarrow{[T]} (\mathbf{L}^{v}(S))^{v} \xrightarrow{[T]} T\end{array} \qquad [T] \leq v \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \mathsf{list}\,L\,N\,C : T}(list)$$

For a soundness proof, all definitions based on either expressions or on types have to be adjusted in order to deal with list constructs. All of these adjustments are reasonably straightforward, and can be found in the PVS formalization. However, the formalization does not yet contain a completely formalized proof; i.e., the list cases are still missing, mainly due to the increased complexity of the proof.

Again, we formulate some examples as lemmas which enables us to formally prove that the typing statements are indeed correct.

Lemma 3. 1. $\mu l.\lambda x.\mathbf{list}\, x\, (\mathbf{inl}\, \square)\, \lambda h.\lambda t.l\, t : (\mathbf{L}^{!}(\alpha^{?}))^{!} \xrightarrow{!} N^{!}$

2. $\mu s.\lambda x.\mathbf{list}\, x\, (\mathbf{inl}\, \square)\, \lambda h.\lambda t.+\, h\, (s\, t) : (\mathbf{L}^{!}(N^{!}))^{!} \xrightarrow{!} N^{!}$

3. $\forall u, v, w \in A, v \leq w, w \leq u : \mu r.\lambda x.\lambda y.\mathbf{list}\, x\, y\, \lambda h.\lambda t.r\, t\, (\mathbf{cons}\, h\, y) :$
$(\mathbf{L}^{v}(\alpha^{u}))^{v} \xrightarrow{v} (\mathbf{L}^{w}(\alpha^{u}))^{w} \xrightarrow{v} (\mathbf{L}^{w}(\alpha^{u}))^{v}$

We use the same representation for natural numbers as in Example 2. The operation + is simply represented by $\perp$, because the only relevant aspect of + is this example us that it is strict in both arguments, and $\perp$ can be typed as a strict binary operation. Example 1 is the *length* function (without addition), and Example 2 the *sum* function. Example 3 is the well-known *reverse* function, that transfers each element of the first list argument to the second argument. One will usually call this function with an empty list as second argument. In that case, all elements of the first list will appear in reverse order in the final result. By using quantified attributes, we obtain a polymorphic strictness typing for reverse. In this typing the difference between the first and the second argument becomes apparent: Even simple hnf-evaluation of an application of reverse will result in the complete evaluation of the spine of the first argument. 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.

9. Discussion of related work

We compared several existing techniques for strictness analysis by giving a brief outline of their main ideas.

In [JI92, Ben93] a non-standard type inference is introduced using conjunction types. The main properties of the system are formulated and proved with respect to a denotational semantics of their language. The difference with our approach is that the strictness information is restricted to traditional head-normal form evaluation only, which, as we argued, hampers modularity. In [HM94] a typing framework is presented focusing on algorithmic aspects, by providing a checking algorithm for a variation of Jensen's system.

The system described by [CDG02] is most similar to ours. For a language resembling the core functional language introduced in Section 2, the authors describe both a strictness and a totality analysis using a non-standard type inference system. The main difference with our approach is that conjunction types are used. In our system the strictness properties of all function arguments

are captured by a single strictness type, whereas the system of [CDG02] requires a conjunction of these properties. The advantage of our approach is that it can be incorporated directly in a standard Hindley-Milner type inference algorithm.

The system introduced by [HH10] is based on *relevance typing*. Similar to our system, and the backwards strictness analysis used in the Glasgow Haskell Compiler [JHH+92], the (evaluation) context in which variables are used determines whether these variables are *relevant* if such a context is evaluated. In [HH10] the emphasis is on exploiting strictness information by defining a transformation replacing ordinary function applications by a more efficient *eager* applications.

Strictness analysis by *abstract interpretation* introduces a non-standard semantics by translating functions into abstract versions over finite domains, notably over finite lattices. The bottom elements of these domains play the role of generic 'undefined' values. Recursive abstract functions are defined by a fixed-point construction. The main property of this alternative interpretation is to yield a decidable approximation of the (in general undecidable) strictness property, even in the higher-order case. This abstraction inevitably leads to information loss. The standard form of abstract interpretation uses the two-point lattice as ground domain. See [Myc81], [BHA85] and [**?**] for more information. Due to the complexity of finding fixed points in abstract domains, abstract interpretation is not very useful for implementing strictness analysis in compilers for functional languages.

Abstract reduction analyzes evaluation of expressions by mimicking reduction on sets of concrete values extended with special elements for undefinedness. This technique approximates ordinary computations closer than for instance abstract interpretation or strictness typing. Rewriting semantics is adjusted by specifying the behaviour of functions on non-standard elements. Abstract reduction sequences may not terminate. A special technique called *reduction path analysis* is used to cut off these sequences in a way that keeps most of the strictness information intact; see [Nöc93], [CHH00]. The main disadvantage of this approach is the lack of modularity; it requires the implementations of the involved functions to perform the analyses effectively.

Strictness typing is a purely syntactic ('intentional') way of deriving strictness information. The resulting strictness information merely depends on the structure of the expressions, particularly on the occurrences of case clauses, and (as in the case of abstract interpretation) not on the computational behaviour on concrete values. The advantage of strictness typing over abstract interpretation is that the first method can be combined with standard typing. For more information, the reader is referred to [LM91].

10. Conclusions and Future Work

In this paper we presented a strictness typing system which is *fully higher-order*. We describe a type derivation system as well as a type inference algorithm. Moreover, the typing system enables the specification of arbitrary evaluation contexts, which is essential for supporting modularity. Like many other meta-theoretical expositions we used De Bruijn indices to represent term

variables. Despite the objections that have been raised against this low-level representation (e.g., See [ABF+05]), we encountered no real issues that significantly hampered our proofs.

We have demonstrated that proof assistants are not only useful in formalizing existing proofs but also to develop new language theoretic concepts. One major concern, however, remains the fact that the construction of a formalized proof remains very time consuming. Compared to a manual construction on paper, the development time using the proposed method probably takes (much more than) three times as long. It is difficult to opt whether this is worth the investment. Although the reader might not learn very much from the formalization itself, it is still useful because in the end it guarantees that the system is indeed fully correct.

10.1. Future work

The formal proof does not yet cover soundness of the entire typing system: the proof work on recursive data types must be extended and completed. Moreover, a proof formalization of the Principal Strictness Typing Theorem is still a compelling challenge. In addition, the type system itself should be considered as a proof of concept, rather than as a system that is directly suited for being incorporated in a standard functional language, like Haskell. For this reason, we have developed a prototype implementation. However, this prototype is still very rudimentary, and will require quite some effort in order to make it ripe for testing and comparing with other strictness analyzers.

References

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[Ben93] Peter Nicholas Benton. *Strictness analysis of lazy functional programs.* Number 309 in Ph.D. Thesis. University of Cambridge, Computer Laboratory, 1993.

[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] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, volume 6, pages 579–612, 1996.

[BS07] Erik Barendsen and Sjaak Smetsers. Strictness analysis via resource typing. In *Reflections on Type Theory, Lambda Calculus, and the Mind*, pages 29–40, Nijmegen, Netherlands, December 2007.

[CDG02] Mario Coppo, Ferruccio Damiani, and Paola Giannini. Strictness, totality, and non-standard-type inference. *Theor. Comput. Sci.*, 272(1-2):69–112, 2002.

[CHH00] David Clark, Chris Hankin, and Sebastian Hunt. Safety of strictness analysis via term graph rewriting. In *Static Analysis, 7th International Symposium, SAS 2000*, LNCS, pages 95–114. Springer, 2000.

[DW90] Kei Davis and Philip Wadler. Backwards strictness analysis: Proved and improved. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, pages 12–30, London, UK, 1990. Springer-Verlag.

[HH10] Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 121–130, New York, NY, USA, 2010. ACM.

[HM94] Chris Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *POPL*, pages 202–212, 1994.

[JHH+92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The glasgow haskell compiler: a technical overview, 1992.

[JI92] T.P. Jensen and Københavns Universitet. Datalogisk Institut. *Abstract interpretation in logical form.* Ph.D. Thesis, DIKU, Datalogisk Institut, Københavns Universitet. DIKU, 1992.

[Kam01] Fairouz Kamareddine. Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. *Logic and Computation*, 11:11–3, 2001.

[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.

[LM91] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher-order lazy languages. In *Proc. of International Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, pages 328–351. Boston, USA, Springer Verlag, LNCS 523, 1991.

[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.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Sme10] Sjaak Smetsers. The syntactic continuity property: A computer verified proof. In Sun-Yuan Hsieh Jixin Ma Zoran Majkic and Khalid S HusainEditors Ibrahiem M M El Emary, editors, *International Conference on Theoretical and Mathematical Foundations of Computer Science (TMFCS10)*, pages 135–142. ISRST, 2010.

[SvE12] Sjaak Smetsers and Marko C. J. D. van Eekelen. Higher-order strictness typing. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[SvEvK09] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2), 2009.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 1–11, New York, NY, USA, 1995. ACM.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[Wal04] David Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–44. MIT Press, 2004.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.