# Effective Strictness Analysis with HORN Constraints

Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann
{keving,pjs,sulzmann}@cs.mu.oz.au

Dept. of Computer Science and Software Engineering,
The University of Melbourne, Victoria 3010, Australia

**Abstract.** We introduce a constraint-based framework for strictness analysis applicable to ML style languages supporting higher-order functions, let-style polymorphism and algebraic datatypes. The analysis produces strictness types for expressions in a program. A strictness type is defined using Boolean constraints. Perhaps surprisingly, the Boolean constraints that arise during analysis are in HORN, which makes the operations on them amenable to efficient implementation.
We have implemented the approach within a highly optimising Haskell compiler (GHC) and give a comparison with the current strictness analyser of GHC.

## 1 Introduction

In lazy functional programming languages expressions are evaluated only if they are needed. A function is strict in an argument if its value is undefined whenever the argument is undefined. Determining which arguments of a function are strict can be used by compilers to improve program performance by replacing call-by-need by call-by value. Hence there has been considerable interest in determining strictness of functions.

Mycroft [9] was the first person to consider strictness analysis of (first-order flat typed) functional programs. His approach to strictness analysis using abstract interpretation maps a function to a strictness function over Booleans. The undefined value is represented by $false$ (which we shall write as $\bot$) while no information (all possible values including the undefined value) is represented by *true* (written $\top$). The translation to a Boolean function preserves just the function's behaviour on undefined values. The result of evaluating this function gives a safe approximation of the real function's behaviour.

So for example the strictness type of $+$, which is strict in both its arguments, is represented by the function

$$+ \; \mathsf{x} \; \mathsf{y} = \mathsf{x} \wedge \mathsf{y}$$

where $\wedge$ is Boolean conjunction.

The strictness of $+$ in its first argument is shown by the fact that $+ \perp \top = \perp$. This means that if the first argument is undefined the result of $+$ must be undefined.

The approach of Mycroft is pleasing in that it simply maps each program construct to a Boolean version. So for example the functions

$$
\begin{aligned}
&\mathsf{f} :: \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Bool} \mapsto \mathbf{Int}\\
&\mathsf{f\ x\ y\ z\ b} = \mathsf{if\ b\ then\ x + y\ else\ z}\\
&\mathsf{g} :: \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Bool} \mapsto \mathbf{Int}\\
&\mathsf{g\ x\ y\ b}\ \ = \mathsf{f\ x\ y\ x\ b}
\end{aligned}
$$

are represented by the Boolean strictness functions

$$
\begin{aligned}
&\mathsf{f} : \mathbf{Bool} \mapsto \mathbf{Bool} \mapsto \mathbf{Bool} \mapsto \mathbf{Bool} \mapsto \mathbf{Bool}\\
&\mathsf{f}\ x\ y\ z\ b = b \wedge ((+\ x\ y) \vee z) = b \wedge ((x \wedge y) \vee z)\\
&\mathsf{g} : \mathbf{Bool} \mapsto \mathbf{Bool} \mapsto \mathbf{Bool} \mapsto \mathbf{Bool}\\
&\mathsf{g}\ x\ y\ b\ \ = \mathsf{f}\ x\ y\ x\ b = b \wedge ((x \wedge y) \vee x) = b \wedge x
\end{aligned}
$$

where $\vee$ is Boolean disjunction. Note that we use the notation $\mapsto$ for function types to differentiate them from Boolean implication $\rightarrow$.

In general a function f is detected to be strict in an argument if f called with that argument $\bot$ and all other arguments $\top$ returns $\bot$. For example f is strict in $b$ but not strict in $x$ since $\mathsf{f}\ \top\ \top\ \top\ \bot = \bot$ and $\mathsf{f}\ \bot\ \top\ \top\ \top = \top$

The natural way to extend Mycroft approach to higher-order functions is to represent their strictness by higher-order Boolean functions and this is indeed what Burn, Hankin and Abramsky [1] defined. Unfortunately in practice manipulating higher-order Boolean function descriptions quickly becomes impractical.

Kuo and Mishra [8] instead suggested using constrained type expressions to express strictness. We take a similar approach in our framework, the strictness of a function is represented by a Boolean *relation* rather than a function. To do so we need to relate the variables of the Boolean relation to the arguments and result of the function, hence we obtain a two part description $(C, \tau)$ the Boolean constraints $C$, and the type annotated with the names of the Boolean variables. For functions we write this in the style of a constrained type $\forall \bar{\delta}.C \Rightarrow \tau$ where $\bar{\delta}$ is a set of free variables not bound in any environment surrounding the function, $\tau$ is the type component and $C$ is the constraint component restricting the set of possible instances of variables $\bar{\delta}$.

For example, we represent f above as

$$\mathsf{f} : \forall\ b\ x\ y\ z\ r.\ (b \wedge ((x \wedge y) \vee z) \leftrightarrow r \Rightarrow x \mapsto y \mapsto z \mapsto b \mapsto r$$

Once we have made the step to relational descriptions, a key realization is that only one direction of the constraint information is required to determine strictness information. Recall that $\top$ indicates no information, hence the statement that $\mathsf{f}\ \bot\ \top\ \top\ \top = \top$ conveys no information, hence it is simply the possibility that $\mathsf{f}\ \bot\ \top\ \top\ \top = \bot$ that illustrates that $f$ is strict in $b$. This information is still captured in the weaker relational description

$$\mathsf{f} : \forall\ b\ x\ y\ z\ r.\ (b \wedge ((x \wedge y) \vee z) \rightarrow r \Rightarrow x \mapsto y \mapsto z \mapsto b \mapsto r$$

There are immediate advantages of keeping only the minimal amount of information:

- The resulting Boolean constraints are in HORN. For example $(b \wedge ((x \wedge y) \vee z) \rightarrow r$ is equivalent to $(b \rightarrow x \rightarrow y \rightarrow r) \wedge (b \rightarrow z \rightarrow r)$.[1] Algorithms for

[1] We shall assume for the remainder of the paper that $\rightarrow$ is right associative and higher precedence than $\wedge$ and usually write Boolean constraints without brackets e.g. $b \rightarrow x \rightarrow y \rightarrow r \wedge b \rightarrow z \rightarrow r$.

manipulating Horn formulae are more efficient than those for manipulating arbitrary Boolean formulae.
– The handling of if-then-else (and other "branching" constructs like case) is considerably simpler since the resulting strictness constraints can simply be conjoined. For example consider the inference of a strictness type for f:

$$\begin{aligned}
\mathsf{x} + \mathsf{y} &: (x \rightarrow y \rightarrow r,\ \ r) \\
\mathsf{z} &: (z \rightarrow r,\ \ r) \\
\mathsf{if\ b\ then\ x + y\ else\ z} &: (b \rightarrow ((x \rightarrow y \rightarrow r) \wedge (z \rightarrow r)),\ \ r) \\
\mathsf{f} &: \forall\ b\ x\ y\ z\ r.\ (b \rightarrow x \rightarrow y \rightarrow r) \wedge (b \rightarrow z \rightarrow r) \Rightarrow x \mapsto y \mapsto z \mapsto b \mapsto r
\end{aligned}$$

where for each subexpression we have written its strictness description on the right-hand side.

For functions with a first-order flat type it is easy to show that our inference algorithms gives the same answers as that of Mycroft [9] and Burn *et al.* [1] (see Theorems 1,3 and 4).

Type-based strictness analyses, such as the approach of Kuo and Mishra and ours, extend to higher-order functions in a natural way. Descriptions just relate the Boolean variables representing the various parts of the type. For example the strictness types of twice f x = f (f x), id x = x and always3 x = 3 are

$$\begin{aligned}
\mathsf{twice} &: \forall \delta_1 \delta_2 \delta_3 \delta_4.\ \delta_3 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_4 \Rightarrow (\delta_1 \mapsto \delta_2) \mapsto \delta_3 \mapsto \delta_4 \\
\mathsf{id} &: \forall \delta_1 \delta_2.\ \delta_1 \rightarrow \delta_2 \Rightarrow \delta_1 \mapsto \delta_2 \\
\mathsf{always3} &: \forall \delta_1 \delta_2.\ \delta_2 \Rightarrow \delta_1 \mapsto \delta_2
\end{aligned}$$

and the inference of strictness types for expressions twice id and twice always3 are straightforwardly obtained by matching arguments and conjoining the resulting constraints. That is

$$\begin{aligned}
\mathsf{twice\ id} &: (\delta_1 \rightarrow \delta_2 \wedge \delta_3 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_4,\ \delta_3 \mapsto \delta_4) = (\delta_3 \rightarrow \delta_4,\ \delta_3 \mapsto \delta_4) \\
\mathsf{twice\ always3} &: (\delta_2 \wedge \delta_3 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_4,\ \delta_3 \mapsto \delta_4) = (\delta_4,\ \delta_3 \mapsto \delta_4)
\end{aligned}$$

Using Boolean constraints for strictness descriptions is advantageous since they are well understood, and operations like conjunction and existential quantification (both used in the above examples) are straightforward and have efficient implementations. Another advantage we claim for using well-understood constraint domains is that extensions of the system are easier to construct. Our approach to strictness analysis handles both let-style polymorphism and algebraic data types.

In this paper we present a strictness analyser for Haskell based on HORN constraints. It is incorporated in the GHC Haskell compiler, and handles the full Haskell 98 language. Although the implementation is still preliminary it is capable of analysing much more accurate strictness information than the current GHC strictness analyser.

The remainder of the paper is organized as follows. In the next section we define our strictness logic and its relation to the abstract interpretation framework of Burn *et al.* [1]. In particular, we prove equivalence on first-order flat types to [1]. In Section 3 we give the inference algorithm for monomorphic programs. In Section 4 we extend the framework to handle let-style polymorphism and (non-recursive) algebraic datatypes. In Section 5 we discuss our implementation in the GHC compiler and compare it with the current strictness analysis. Finally in Section 6 we discuss related work and conclude.

## 2 Strictness logic

For the moment, we assume that the underlying language is monomorphic:

| | |
|---|---|
| **Types** | $t ::= \mathbf{Int} \mid \mathbf{Bool} \mid t \mapsto t$ |
| **Expressions** | $e ::= x \mid \lambda x.e \mid e\,e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{fix}\ x\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$ |

We will in later sections show how to incorporate ML-style polymorphism and algebraic data types.

We define a strictness logic in terms of a constraint-based type system. Strictness properties of expressions are specified by typing judgments which are of the form $C, \Gamma\, e : \eta$ where $C$ is a strictness constraint, $\Gamma$ is a strictness type environment assigning strictness types to the free variables of $e$ and $\eta$ is $e$'s strictness type. We always assume that expression $e$ is well-typed in the underlying type system.

### Strictness types

The structure of strictness types reflects the structure of the underlying type system.

| | |
|---|---|
| **Annotations** | $b ::= \delta \mid \bot \mid \top$ |
| **Strictness Types** | $\tau ::= b \mid \tau \mapsto \tau$ |
| **Type Schemes** | $\eta ::= \tau \mid \forall \bar{\delta}.C \Rightarrow \tau$ |

We write $\bar{\delta} = fv(\eta)$ to refer to the free variables in a strictness type $\eta$. The constraint component $C$ will be described shortly.

### Strictness constraints

Strictness constraints are expressed in the class HORN of propositional formulas. In addition, we find conjunction and existential quantification. The language of constraints is as follows:

$$C ::= \bot \mid \top \mid \delta_1 \to \ldots \to \delta_n \mid C \wedge C \mid \exists \bar{\delta} C$$

where $\bot$ denotes $false$ and $\top$ denotes $true$. We write $C \models D$ to denote model-theoretic entailment among constraints $C$ and $D$.

Structural strictness constraints $(\tau \leq_s \tau')$ are translated as follows:

$$\begin{aligned} [\![(\tau_1 \mapsto \tau_1' \leq_s \tau_2 \mapsto \tau_2')]\!] &= [\![(\tau_1' \leq_s \tau_2')]\!] \wedge [\![(\tau_2 \leq_s \tau_1)]\!] \\ [\![(\delta_1 \leq_s \delta_2)]\!] &= \delta_1 \to \delta_2 \end{aligned}$$

Note that types $\tau$ and $\tau'$ in structural constraints $(\tau \leq_s \tau')$ must always be of the same shape. This condition is enforced by requiring expressions to be well-typed in the underlying type system.

$$
(\mathrm{Var})\ \frac{(x:\eta)\in\Gamma}{C,\Gamma\ x:\eta}
\qquad
(\mathrm{Sub})\ \frac{C,\Gamma\ e:\tau_2 \quad C\models[\![(\tau_2\leq_s\tau_1)]\!]}{C,\Gamma\ e:\tau_1}
$$

$$
(\mathrm{Abs})\ \frac{C,\Gamma_x.x:\tau_1\ e:\tau_2}{C,\Gamma_x\ \lambda x.e:\tau_1\mapsto\tau_2}
\qquad
(\mathrm{App})\ \frac{\begin{array}{c}C,\Gamma\ e_1:(\tau_1\mapsto\tau_2)\\ C,\Gamma\ e_2:\tau_1\end{array}}{C,\Gamma\ e_1e_2:\tau_2}
$$

$$
(\mathrm{Let})\ \frac{\begin{array}{c}C,\Gamma_x\ e_1:\eta\\ C,\Gamma_x.x:\eta\ e_2:\tau\end{array}}{C,\Gamma_x\ \mathsf{let}\ x=e_1\ \mathsf{in}\ e_2:\tau}
\qquad
(\exists\mathrm{I})\ \frac{\begin{array}{c}C,\Gamma\ e:\tau\\ \bar\delta\notin \mathit{fv}(\Gamma,\tau)\end{array}}{\exists\bar\delta C,\Gamma\ e:\tau}
$$

$$
(\forall\mathrm{I})\ \frac{\begin{array}{c}C\wedge D,\Gamma\ e:\tau\\ \bar\delta=\mathit{fv}(D,\tau)\backslash \mathit{fv}(\Gamma,C)\end{array}}{C\wedge\exists\bar\delta D,\Gamma\ e:\forall\bar\delta.D\Rightarrow\tau}
\qquad
(\forall\mathrm{E})\ \frac{\begin{array}{c}C,\Gamma\ x:\forall\bar\delta.D\Rightarrow\tau\\ C\models[\bar b/\bar\delta]D\end{array}}{C,\Gamma\ x:[\bar b/\bar\delta]\tau}
$$

$$
(\mathrm{If})\ \frac{\begin{array}{c}C,\Gamma\ e:\delta\\ C,\Gamma\ e_1:\tau_1\quad C,\Gamma\ e_2:\tau_2\\ C\models\delta\rightarrow[\![(\tau_1\leq_s\tau)\wedge(\tau_2\leq_s\tau)]\!]\end{array}}{C,\Gamma\ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau}
\qquad
(\mathrm{Fix})\ \frac{C,\Gamma_x.x:\eta\ e:\eta}{C,\Gamma_x\ \mathsf{fix}\ x\ \mathsf{in}\ e:\eta}
$$

**Fig. 1.** Strictness Typing Rules

### Typing rules

Figure 1 defines the strictness logic. Most of its rules are straightforward. $\Gamma_x$ denotes the environment obtained from $\Gamma$ by excluding variable $x$. Rule (If) handles conditional expressions. Note that well-typing in the underlying system ensures that expression $e$ is of type **Bool** and expressions $e_1$ and $e_2$ share the same underlying type. Therefore, their strictness types are of the same shape. The rule states that if the condition $e$ is undefined, i.e. $C \models \delta \rightarrow \bot$, then the resulting type $\tau$ is unconstrained. If we have no information about the condition, i.e. $C \models \delta \rightarrow \top$, then we build the least upper bound of the strictness descriptions of $e_1$ and $e_2$. Note that the constraint $\delta \rightarrow [\![(\tau_1 \leq_s \tau) \wedge (\tau_2 \leq_s \tau)]\!]$ is in HORN. Rules (∀I) and (∀E) handle construction and elimination of polyvariant strictness types. Note that for now we only allow for polymorphism in the annotation variables (polyvariance). In Section 4, we show how to allow in addition for polymorphism in the underlying language. Rule (∃I) allows unnecessary strictness type variables in the constraint component of a typing judgement to be hidden.

### Relationship to abstract interpretation

We relate our strictness logic to the strictness analysis by abstract interpretation of Burn *et al.* [1]. We show that in case of first-order programs our strictness logic is equivalent to [1]. In case of higher-order programs we are sound with respect to [1] which establishes correctness of our analysis.

Strictness analysis by abstract interpretation interprets well-typed expressions in the complete partial order $\mathcal{V}$ where $\mathcal{V}$ is the least solution of the equation

$$\mathcal{V} = \mathbf{2} + \mathcal{V} \mapsto \mathcal{V}$$

where $\mathbf{2}$ is the two–point lattice $\{\bot, \top\}$ such that $\bot \leq \top$.

The abstraction of an expression $e$ of type $\mathbf{Bool} \mapsto \mathbf{Int} \mapsto \mathbf{Int}$ would then be a function $[\![e]\!] :: \mathbf{2} \mapsto \mathbf{2} \mapsto \mathbf{2}$. The function $[\![\cdot]\!]$ on expressions is defined as follows (where $\rho$ is a variable mapping from variables to values):

$$\begin{array}{ll}
[\![x]\!]\rho & = \rho(x) \\
[\![\lambda u.e]\!]\rho & = \lambda v.[\![e]\!]\rho[u := v] \\
[\![e\,e']\!]\rho & = [\![e]\!]\rho([\![e']\!]\rho) \\
[\![\mathsf{let}\, x = e \,\mathsf{in}\, e']\!]\rho & = [\![e']\!]\rho[x := [\![e]\!]\rho] \\
[\![\mathsf{fix}\ x \,\mathsf{in}\ e]\!]\rho & = \mathit{fix}(\lambda v.[\![e]\!]\rho[x := v]) \\
[\![\mathsf{if}\ e \,\mathsf{then}\ e_1 \,\mathsf{else}\ e_2]\!]\rho & = \begin{cases} \bot & \text{if } [\![e]\!]\rho = \bot \\ [\![e_1]\!]\rho \sqcup [\![e_2]\!]\rho & \text{otherwise} \end{cases}
\end{array}$$

The meaning of strictness types is defined in terms of ideals in $\mathcal{V}$. Ideals are non-empty, downward-closed and limit-closed subsets of $\mathcal{V}$.

The function $[\![\cdot]\!]$ on closed strictness types is defined below. We say a type $\tau$ is a *monotype* if $\mathit{fv}(\tau) = \emptyset$. We let $\mu$ range over monotypes and $\nu$ over $\bot$ and $\top$.

$$\begin{array}{ll}
[\![\bot]\!] & = \{\bot\} \\
[\![\top]\!] & = \{\bot, \top\} \\
[\![\mu \mapsto \mu']\!] & = \{f \in \mathcal{V} \mapsto \mathcal{V} \mid v \in [\![\mu]\!] \Rightarrow f\,v \in [\![\mu']\!]\} \\
[\![\forall\bar{\delta}.C \Rightarrow \tau]\!] & = \bigcap \{\, [\![[\bar{\nu}/\bar{\delta}]\tau]\!] \mid\ \models [\bar{\nu}/\bar{\delta}]C \,\}
\end{array}$$

Note that $[\![\eta]\!]$ is an ideal for a closed strictness type $\eta$. The strictness type system is sound, i.e. every satisfiable constraint has a monotype solution. The $(\leq_s)$ relation is coherent, i.e. if a type $\tau$ subsumes a type $\tau'$, written $\models (\tau \leq_s \tau')$, then the denotation of $\tau$ in the ideal model is a subset of the denotation of $\tau'$.

Our first result concerns only first-order programs. We can show that our HORN-based strictness system is equivalent to Burn *et al.*'s abstract interpretation approach in case of first-order programs.

In a first step, we observe that Burn *et al.*'s strictness descriptions for first-order functions, which in fact are Boolean functions, can be viewed as constrained types in the Boolean domain. For example the function $+$ is described by the Boolean function $\delta_1 \mapsto \delta_2 \mapsto (\delta_1 \wedge \delta_2)$ which in turn can be viewed as $\forall\bar{\delta}.(\delta_1 \wedge \delta_2) \leftrightarrow \delta_3 \Rightarrow \delta_1 \mapsto \delta_2 \mapsto \delta_3$. We can be more specific. Every first-order function $f$ can be described by $\forall\bar{\delta}.P \leftrightarrow \delta_n \Rightarrow \delta_1 \mapsto \ldots \delta_n$ where $P$ is a positive boolean formulae where the class of positive boolean formulas (POS) is defined as:

$$P ::= \bot \mid \top \mid \delta \mid P \wedge P \mid P \vee P \mid \exists\bar{\delta}.P$$

We further observe that we can weaken the strictness description of first-order functions without losing any information.

**Lemma 1 (Equivalence between HORN and POS Types).** *Let $P$ be a boolean expression in POS. Let $\eta = \forall\bar{\delta}.P \leftrightarrow \delta_n \Rightarrow \delta_1 \mapsto \ldots \mapsto \delta_n$ be a polymorphic strictness type such that $\mathit{fv}(\eta) = \emptyset$. Then, $P \to \delta_n$ is in HORN and*

$$[\![\forall\bar{\delta}.P \leftrightarrow \delta_n \Rightarrow \delta_1 \mapsto \ldots \mapsto \delta_n]\!] = [\![\forall\bar{\delta}.P \to \delta_n \Rightarrow \delta_1 \mapsto \ldots \mapsto \delta_n]\!]$$

This allows us to establish the result that HORN-based inference is sound and complete wrt. Burn *et al.*'s abstract interpretation approach in case of first-order programs.

A variable environment $\rho$ *models* a closed typing environment $\Gamma$, written $\rho \models \Gamma$, if for all $x : \eta \in \Gamma$, $\rho(x) \in [\![\eta]\!]$. We write $\Gamma \models e : I$ iff for all $\rho$ where $\rho \models \Gamma$ we have that $[\![e]\!]\rho \in I$ where $I$ is an ideal.

**Theorem 1 (Soundness and Completeness of First-Order Programs).** *Let $e$ be a first-order program, $\eta$ a strictness type, $\Gamma$ a strictness environment such that $fv(\Gamma, \eta) = \emptyset$. Then, $true, \Gamma\, e : \eta$ iff $\Gamma \models e : I$ where $I = [\![\eta]\!]$.*

In case of higher-order programs we can just say that we are sound with respect to Burn *et al.*.

**Theorem 2 (Soundness of Higher-Order Programs).** *Let $C, \Gamma\, e : \eta$ be a valid typing judgment. Let $\phi$ be a substitution such that $\phi\Gamma$ and $\phi\eta$ are closed and $\models \phi C$. Let $\rho$ be a variable environment such that $\rho \models \phi\Gamma$. Then $[\![e]\!]\rho \in [\![\phi\eta]\!]$.*

## 3 Inference

We assume that we are given a well-typed program, each subexpression annotated with its underlying type. We write $(e :: t)$ to denote that expression $e$ has the underlying type $t$.

Strictness inference computes the missing strictness information for a type-annotated expression $(e :: t)$. In a first step we compute the shape of $e$'s strictness type, given the underlying type $t$. To this purpose, we introduce judgments of the form $t\,\tau$ which states that $\tau$ is a strictness type with the same shape as $t$. The rules are as follows:

$$(\mathbf{Bool})\,\frac{\delta \text{ is fresh}}{\mathbf{Bool}\,\delta} \quad (\mathbf{Int})\,\frac{\delta \text{ is fresh}}{\mathbf{Int}\,\delta} \quad (\text{Arrow})\,\frac{t_1\,\tau_1 \quad t_2\,\tau_2}{t_1 \mapsto t_2\,\tau_1 \mapsto \tau_2}$$

In Figure 2 we define the inference algorithm which is formulated as a deduction system over clauses of the form

$$\Gamma, e :: t \vdash_{inf} (C, \tau)$$

with a strictness environment $\Gamma$ and a type-annotated expression $e :: t$ as input, and a strictness constraint $C$ and a strictness type $\tau$ as output. We note that an algorithm in style of $\mathcal{W}$ can be straightforwardly derived from the given specification.

All rules are syntax-directed except rule ($\exists$Intro). This rule is justified by the corresponding rule in the logical system and can be applied at any stage of the inference process. Rule (Var-$\lambda$) handles lambda-bound variables whereas rule (Var) handles instantiation of let-bound variables. Rule (Let) introduces annotation polymorphism. We define a generalisation function giving the generalised type scheme and the generalised constraint. Let $C$ be a constraint, $\Gamma$ a type environment, and $\tau$ a type. Then

$$gen(C, \Gamma, \tau) = (\exists\bar{\delta}.C, \forall\bar{\delta}.C \Rightarrow \tau)$$

$$\text{(Var)}\ \frac{(x : \forall\bar{\delta}.C \Rightarrow \tau) \in \Gamma \quad \bar{\delta}' \text{ new}}{\Gamma, x :: t \vdash_{inf} ([\![ [\bar{\delta}'/\bar{\delta}]C ]\!], [\bar{\delta}'/\bar{\delta}]\tau)} \qquad \text{(Var–}\lambda\text{)}\ \frac{\begin{array}{c}(x : \tau) \in \Gamma \quad t\,\tau' \\ C = [\![(\tau \leq_s \tau')]\!]\end{array}}{\Gamma, x :: t \vdash_{inf} (C, \tau')}$$

$$\text{(Abs)}\ \frac{\begin{array}{c} t_1\,\tau_1 \\ \Gamma_x.x : \tau_1, e :: t_2 \vdash_{inf} (C, \tau_2)\end{array}}{\Gamma_x, \lambda x.e :: t_1 \to t_2 \vdash_{inf} (C, \tau_1 \mapsto \tau_2)} \quad \text{(App)}\ \frac{\begin{array}{c}\Gamma, e_1 :: t_1 \vdash_{inf} (C_1, \tau_1) \\ \Gamma, e_2 :: t_2 \vdash_{inf} (C_2, \tau_2) \\ t_3\,\tau_3 \\ C = C_1 \wedge C_2 \wedge [\![(\tau_1 \leq_s \tau_2 \mapsto \tau_3)]\!]\end{array}}{\Gamma, ((e_1 :: t_1)(e_2 :: t_2)) :: t_3 \vdash_{inf} (C, \tau_3)}$$

$$\text{(Let)}\ \frac{\begin{array}{c}\Gamma_x, e_1 :: t_1 \vdash_{inf} (C_1, \tau_1) \\ gen(C_1, \Gamma_x, \tau_1) = (C_0, \eta_0) \\ \Gamma_x.x : \eta_0, e_2 :: t_2 \vdash_{inf} (C_2, \tau_2) \\ C = C_0 \wedge C_2\end{array}}{\Gamma_x, (\mathsf{let}\ x = (e_1 :: t_1)\ \mathsf{in}\ e_2) :: t_2 \vdash_{inf} (C, \tau_2)} \qquad (\exists\text{Intro})\ \frac{\begin{array}{c}\Gamma, e :: t \vdash_{inf} (C, \tau) \\ \bar{\delta} = fv(C) \backslash fv(\Gamma, \tau)\end{array}}{\Gamma, e :: t \vdash_{inf} (\exists\bar{\delta}C, \tau)}$$

$$\text{(If)}\ \frac{\begin{array}{c}\Gamma, e :: \mathbf{Bool} \vdash_{inf} (C_0, \delta) \quad t\,\tau \\ \Gamma, e_1 :: t \vdash_{inf} (C_1, \tau_1) \quad \Gamma, e_2 :: t \vdash_{inf} (C_2, \tau_2) \\ C = C_0 \wedge C_1 \wedge C_2 \wedge (\delta \to [\![(\tau_1 \leq_s \tau) \wedge (\tau_2 \leq_s \tau)]\!])\end{array}}{\Gamma, (\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 :: t) \vdash_{inf} (C, \tau)}$$

$$\text{(Fix)}\ \frac{\begin{array}{l} t\,\tau \quad \bar{\delta} = fv(\tau) \\ \eta_0 = \forall\bar{\delta}.true \Rightarrow \tau \\ \mathcal{F}(\Gamma_x.x : \eta_0, e :: t) = (C, \tau)\end{array}}{\Gamma_x, (\mathsf{fix}\ x :: t\ \mathsf{in}\ e) :: t \vdash_{inf} (C, \tau)}$$

**Fig. 2.** Strictness Inference

where $\bar{\delta} = fv(C, \tau) \backslash fv(\Gamma)$. Note that we push the whole constraint $C$ into the type scheme (we could be more efficient by pushing in only the *affected* constraints).

In rule (If), the constraint $\delta \to [\![(\tau_1 \leq_s \tau) \wedge (\tau_2 \leq_s \tau)]\!]$, where $[\![(\tau_1 \leq_s \tau) \wedge (\tau_2 \leq_s \tau)]\!] = (\delta_1 \to \delta_1') \wedge \ldots \wedge (\delta_n \to \delta_n')$, translates to $\delta \to ((\delta_1 \to \delta_1') \wedge \ldots \wedge (\delta_n \to \delta_n'))$ which is equivalent to $\delta \to \delta_1 \to \delta_1' \wedge \ldots \wedge \delta \to \delta_n \to \delta_n'$. Hence, all constraints we generate are in HORN.

With (Fix), we follow Dussart, Henglein and Mossin [2], performing a Kleene-Mycroft iteration until a fixpoint is found. We define $(\forall\bar{\delta}_1.C_1 \Rightarrow \tau_1) \leq (\forall\bar{\delta}_2.C_2 \Rightarrow \tau_2)$ iff $C_2 \models \exists\bar{\delta}_1.(C_1 \wedge [\![(\tau_1 \leq_s \tau_2)]\!])$. W.l.o.g. we assume there are no name clashes between $\bar{\delta}_1$ and $\bar{\delta}_2$. Define

$$\begin{array}{ll} \mathcal{F}(\Gamma_x.x : \eta_i, e :: t) & \\ = \mathcal{F}(\Gamma_x.x : \eta_{i+1}, e :: t) & \text{if } \eta_i < \eta_{i+1} \\ = (C, \tau) & \text{if } \eta_i = \eta_{i+1} \end{array}$$

where $\Gamma_x.x : \eta_i, e :: t \vdash_{inf} (C, \tau)$ and $(\cdot, \eta_{i+1}) = gen(C, \Gamma_x.x : \eta_i, \tau)$. Clearly the sequence $\eta_o \leq \ldots \leq \eta_i \leq \ldots$ is finite.

*Example 1.* Let $t = \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Int}$. Consider the inference for $\mathsf{km} :: t$.

```
km x y z = if z == 1 then x - y else km y x (z - 1)
```

Let $\Gamma = \{\mathsf{x} : \delta_x, \mathsf{y} : \delta_y, \mathsf{z} : \delta_z, \mathsf{km} : \forall \delta_1\delta_2\delta_3\delta_4.\ \mathit{true} \Rightarrow \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4\}$ plus strictness types for 1, $==$ and $-$.

$$
\begin{array}{ll}
 & \Gamma, 1 :: \mathbf{Int} \vdash_{inf} (\delta_5, \delta_5) \\
\text{(App)} & \Gamma, \mathsf{z == 1} :: \mathbf{Bool} \vdash_{inf} (\delta_5 \to \delta_z \to \delta_6 \wedge \delta_5, \delta_6) \\
(\exists\text{Intro}) & \Gamma, \mathsf{z == 1} :: \mathbf{Bool} \vdash_{inf} (\delta_z \to \delta_6, \delta_6) \\
\text{(App)}(\exists\text{Intro}) & \Gamma, \mathsf{z\ \text{-}\ 1} :: \mathbf{Int} \vdash_{inf} (\delta_z \to \delta_7, \delta_7) \\
\text{(App)}(\exists\text{Intro}) & \Gamma, \mathsf{x\ \text{-}\ y} :: \mathbf{Int} \vdash_{inf} (\delta_x \to \delta_y \to \delta_8, \delta_8) \\
\text{(Var)} & \Gamma, \mathsf{km} :: t \vdash_{inf} (\mathit{true}, \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4)
\end{array}
$$

$$
\begin{array}{lll}
\text{(App)} & \Gamma, \mathsf{km\ y\ x\ (z\ \text{-}\ 1)} :: \mathbf{Int} \vdash_{inf} & (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_7 \to \delta_3 \wedge \delta_4 \to \delta_9 \wedge \delta_z \to \delta_7, \delta_9) \\
\text{(If)} & \Gamma, \mathsf{if} \ldots :: \mathbf{Int} \vdash_{inf} & (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_7 \to \delta_3 \wedge \delta_4 \to \delta_9 \wedge \delta_z \to \delta_7 \wedge \\
 & & \delta_z \to \delta_6 \wedge \delta_x \to \delta_y \to \delta_8 \wedge \delta_6 \to \delta_8 \to \delta_r \wedge \delta_6 \to \delta_9 \to \delta_r, \delta_r) \\
(\exists\text{Intro}) & \Gamma, \mathsf{if} \ldots :: \mathbf{Int} \vdash_{inf} & (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_z \to \delta_3 \wedge \delta_4 \to \delta_z \to \delta_r \wedge \\
 & & \delta_x \to \delta_y \to \delta_z \to \delta_r, \delta_r) \\
\text{(Abs)}\times 3 & \Gamma, \mathsf{km} :: t \vdash_{inf} & (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_z \to \delta_3 \wedge \delta_4 \to \delta_z \to \delta_r \wedge \\
 & & \delta_x \to \delta_y \to \delta_z \to \delta_r, \delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r) \\
(\exists\text{Intro}) & \Gamma, \mathsf{km} :: t \vdash_{inf} & (\delta_x \to \delta_y \to \delta_z \to \delta_r, \delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r)
\end{array}
$$

In the last step we obtain the first approximation of km in the fixpoint iteration. We can evaluate the next iteration by adding the constraint $\delta_1 \to \delta_2 \to \delta_3 \to \delta_4$ to the constraints from the (Var) step onward. This time in the last step the elimination of variables $\{\delta_1, \delta_2, \delta_3, \delta_4\}$ results in the same description. Hence the the fixpoint is reached and we have

$$\mathsf{km} : \forall \delta_x\delta_y\delta_z\delta_r.\ \delta_x \to \delta_y \to \delta_z \to \delta_r \Rightarrow \delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r$$

That is km is strict in all arguments.

We can prove soundness and completeness results for well-typed programs:

Soundness states that every deduction derived by the inference system can also be derived in the logical system.

**Theorem 3 (Soundness of Inference).** *Let $\Gamma, e :: t \vdash_{inf} (C, \tau)$. Then $C, \Gamma\, e : \tau$.*

Completeness states that every deduction derivable in the logical system is subsumed by a deduction in the inference system.

**Theorem 4 (Completeness of Inference).** *Let $C, \Gamma\, e : \forall \bar{\delta}_1.C_1 \Rightarrow \tau_1$ and $(e :: t)$ be the type-annotated version of expression $e$ such that $\tau_1$ is of shape $t$. Then $\Gamma, e :: t \vdash_{inf} (C_2, \tau_2)$ for some $C_2, \tau_2, \bar{\delta}_2$ such that $\bar{\delta}_2 = \mathit{fv}(C_2, \tau_2) \backslash \mathit{fv}(\Gamma)$ and $C \wedge C_1 \models \exists \bar{\delta}_2.(C_2 \wedge [\![(\tau_2 \leq_s \tau_1)]\!])$.*

### 3.1 Constraint-based fixpoints

If we are willing to give up some accuracy in the presence of polymorphic recursion in strictness types then we can replace the (Fix) rule by a simpler rule which just ensures that the constraints generated will be above the least fixpoint. This is similar to the (FIX) rule of Kuo and Mishra [8] (they only gave a constraint based fixpoint rule, no iterative fixpoint rule).

The following rule forces all recursive invocations to have the same strictness type which must be a super-type of the strictness type of the recursively defined expression.

$$\text{(FixC)}\ \frac{t\,\tau' \quad \Gamma_x.x : \tau', e :: t \vdash_{inf} (C_1, \tau) \quad t\,\tau'' \qquad C = C_1 \wedge [\![(\tau \leq_s \tau')]\!] \wedge [\![(\tau \leq_s \tau'')]\!]}{\Gamma_x, (\mathsf{fix}\ x :: t\,\mathsf{in}\ e) :: t \vdash_{inf} (C, \tau'')}$$

The "shortcut" of stipulating $(\tau \leq_s \tau')$ may have the unfortunate side-effect of introducing constraints amongst the arguments to a function. An example of this phenomenon appears in Example 2 below. A simple way of eliminating such constraints is to couch the result in terms of a fresh binding-time type $\tau''$, with the constraint $(\tau \leq_s \tau'')$.

This approach may lose accuracy, since it forces the strictness of the arguments in the recursive call to be the same as the strictness for the overall function. It finds a correct but inaccurate strictness type for the function.

*Example 2.* We illustrate the constraint-based fixpoint rule on km starting from the second last inference step of Example 1. Let $\tau' = \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4$ and $\tau'' = \delta''_x \mapsto \delta''_y \mapsto \delta''_z \mapsto \delta''_r$. The (FixC) rule proceeds by adding the constraints $[\![(\delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r \leq_s \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4)]\!]$ and $[\![(\delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r \leq_s \delta''_x \mapsto \delta''_y \mapsto \delta''_z \mapsto \delta''_r)]\!]$ to those inferred in the second last inference step of Example 1 as follows:

$$\begin{array}{ll}
& \Gamma, \mathsf{km} :: t \vdash_{inf} (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_z \to \delta_3 \wedge \delta_4 \to \delta_z \to \delta_r \wedge \\
& \qquad \delta_x \to \delta_y \to \delta_z \to \delta_r, \delta_x \mapsto \delta_y \mapsto \delta_z \mapsto \delta_r) \\
\text{(FixC)} & \Gamma_{km}, \mathsf{fix\ km\ in}\ \ldots :: t \vdash_{inf} (\delta_y \to \delta_1 \wedge \delta_x \to \delta_2 \wedge \delta_z \to \delta_3 \wedge \delta_4 \to \delta_z \to \delta_r \wedge \\
& \qquad \delta_x \to \delta_y \to \delta_z \to \delta_r \wedge \delta_1 \to \delta_x \wedge \delta_2 \to \delta_y \wedge \delta_3 \to \delta_z \wedge \delta_4 \to \delta_r \wedge \\
& \qquad \delta''_x \to \delta_x \wedge \delta''_y \to \delta_y \wedge \delta''_z \to \delta_z \wedge \delta_r \to \delta''_r, \delta''_x \mapsto \delta''_y \mapsto \delta''_z \mapsto \delta''_r) \\
(\exists\text{Int}) & \Gamma_{km}, \mathsf{fix\ km\ in}\ \ldots :: t \vdash_{inf} (\delta''_x \to \delta''_y \to \delta''_z \to \delta''_r, \delta''_x \mapsto \delta''_y \mapsto \delta''_z \mapsto \delta''_r)
\end{array}$$

Note the spurious consequence $\delta_x \leftrightarrow \delta_y$ that arises on the variables of $\tau$ in addition to the correct answer $\delta_x \to \delta_y \to \delta_z \to \delta_r$. This is eliminated when we couch the results in terms of fresh type $\tau''$.

For this example, the constraint-based fixpoint leads to no loss of accuracy since the strictness of the two arguments $x$ and $y$ which were forced to be equal were already equal. Notice that this is an example where the approach of Kuo and Mishra [8] loses accuracy.

The rule is not complete wrt. the logic in Section 2. The result using (FixC) for the function h x y b = if b then x else h x (h y x b) b is

$$\mathsf{h} : \forall\ x\ y\ b.\ b \to x \to r \wedge b \to y \to r \Rightarrow x \mapsto y \mapsto b \mapsto r$$

rather than $\mathsf{h} : \forall\ x\ y\ b.\ b \to x \to r \Rightarrow x \mapsto y \mapsto b \mapsto r$. However, the (FixC) rule is sound.

**Theorem 5 (Soundness of FixC).** *Let $\Gamma, e :: t \vdash_{inf} (C, \tau)$ using (FixC). Then $C, \Gamma\, e : \tau$.*

## 4 Polymorphism and algebraic data types

Any realistic program analysis needs to be able to handle additional language features such as polymorphism and algebraic data types. We first extend our strictness analysis to ML-style polymorphism. Thereafter, we give a description of our handling of algebraic data types.

### 4.1 Polymorphism

In previous work [4], we have already described how to extend Dussart *et al*'s binding-time analysis [2] to programs which are polymorphic in their underlying types. Fortunately, most of the methods and concepts introduced for the binding-time analysis of polymorphic programs carry over to the strictness analysis described in this paper.

*Example 3.* Consider the following polymorphic program:

$$\begin{array}{l} \mathsf{sel} :: \forall \bar{\alpha}.\mathbf{Bool} \mapsto (\alpha_1 \mapsto \alpha_2) \mapsto (\alpha_1 \mapsto \alpha_2) \mapsto \alpha_1 \mapsto \alpha_2 \\ \mathsf{sel\ b\ f\ g\ x} = \mathsf{if\ b\ then\ f\ x\ else\ g\ x} \end{array}$$

which either applies function f or function g to value x, depending on the boolean expression b.

The strictness description of sel is as follows:

$$\forall \delta, \bar{\beta}. \begin{array}{l} \beta_{13} \rightarrow \beta_{11} \wedge \beta_{13} \rightarrow \beta_{12} \wedge \\ \delta \rightarrow \beta_{21} \rightarrow \beta_{23} \wedge \delta \rightarrow \beta_{22} \rightarrow \beta_{23} \end{array} \Rightarrow \delta \mapsto (\beta_{11} \mapsto \beta_{21}) \mapsto (\beta_{12} \mapsto \beta_{22}) \mapsto \beta_{13} \mapsto \beta_{23}$$

We use the convention that variables $\beta$ refer to strictness variables which correspond to a polymorphic variable in the underlying system. For example, strictness variables $\beta_{11}$, $\beta_{12}$ and $\beta_{13}$ correspond to variable $\alpha_1$. Note that although we still use HORN constraints, constraints of the form $\beta_{13} \rightarrow \beta_{11}$ now express structural relationships between instances of $\beta_{13}$ and $\beta_{11}$. In a system without Boolean constraints, we would write $(\beta_{13} \leq_s \beta_{11})$ instead. The advantage of using Booleans is that we do not need to introduce any new primitive constraints. For example, the constraint $\delta \rightarrow \beta_{21} \rightarrow \beta_{23}$ states that if $\delta = \top$ then $(\beta_{21} \leq_s \beta_{23})$ must hold, otherwise there are no constraints imposed on the relationship between instances of $\beta_{21}$ and $\beta_{23}$.

Assume that we apply function sel in the following context:

$$\begin{array}{lll} \mathsf{sel\ b\ f\ g\ e} & :: \mathbf{Int} \mapsto \mathbf{Int} & : (C, \tau) \\ \mathsf{b} & :: \mathbf{Bool} & : (C_b, \delta_b) \\ \mathsf{f} & :: (\mathbf{Int} \mapsto \mathbf{Int}) \mapsto (\mathbf{Int} \mapsto \mathbf{Int}) & : (C_f, (\delta_{f1} \mapsto \delta_{f2}) \mapsto (\delta_{f3} \mapsto \delta_{f4})) \\ \mathsf{g} & :: (\mathbf{Int} \mapsto \mathbf{Int}) \mapsto (\mathbf{Int} \mapsto \mathbf{Int}) & : (C_g, (\delta_{g1} \mapsto \delta_{g2}) \mapsto (\delta_{g3} \mapsto \delta_{g4})) \\ \mathsf{e} & :: \mathbf{Int} \mapsto \mathbf{Int} & : (C_e, \delta_{e1} \mapsto \delta_{e2}) \end{array}$$

where we have type-annotated all expressions and written all strictness descriptions on the right-hand side.

We obtain the strictness description $(C, \tau)$ of expression sel b f g e as follows. We perform the following substitutions on the type component of sel

$$\begin{array}{ll} \beta_{11} = \delta_{f1} \mapsto \delta_{f2} & \beta_{21} = \delta_{f3} \mapsto \delta_{f4} \\ \beta_{12} = \delta_{g1} \mapsto \delta_{g2} & \beta_{22} = \delta_{g3} \mapsto \delta_{g4} \\ \beta_{13} = \delta_{e1} \mapsto \delta_{e2} & \beta_{23} = \delta_1 \mapsto \delta_2 \end{array}$$

which shows that $\tau = \delta_1 \mapsto \delta_2$. On the constraint side, it must hold that all relations which hold for polymorphic strictness variables carry over to their instances. For example, the constraint $\beta_{13} \to \beta_{11}$ generates the constraint $[\![(\delta_{e1} \mapsto \delta_{e2} \leq_s \delta_{f1} \mapsto \delta_{f2})]\!]$ and $\delta \to \beta_{21} \to \beta_{23}$ generates the constraint $\delta \to [\![(\delta_{f3} \mapsto \delta_{f4} \leq_s \delta_1 \mapsto \delta_2)]\!]$. We find

$$C = \begin{array}{l} \delta_{f1} \to \delta_{e1} \wedge \delta_{e2} \to \delta_{f2} \wedge \delta_{g1} \to \delta_{e1} \wedge \delta_{e2} \to \delta_{g2} \wedge \\ \delta \to \delta_1 \to \delta_{f3} \wedge \delta \to \delta_{f4} \to \delta_2 \wedge \\ \delta \to \delta_1 \to \delta_{g3} \wedge \delta \to \delta_{g4} \to \delta_2 \wedge \\ C_b \wedge C_f \wedge C_g \wedge \delta_b \to \delta \end{array}$$

In general, the method for building an instance of a strictness type scheme is as follows. We first relate polymorphic strictness variables to polymorphic variables in the underlying system. To this purpose, we introduce judgments $\Delta\,\tau : t$ where $\Delta$ is a shape environment associating strictness variables $\beta$ to underlying variables $\alpha$. We assume that the underlying type language is now as follows:

$$\begin{array}{ll} \textbf{Types} & t ::= \alpha \mid \textbf{Int} \mid \textbf{Bool} \mid t \to t \\ \textbf{Type Schemes} & \sigma ::= t \mid \forall \bar{\alpha}.t \end{array}$$

Judgments $\Delta\,\tau : t$ are derived as follows:

$$(\textbf{Bool}_\delta)\ \Delta\,\delta : \textbf{Bool} \qquad (\textbf{Bool}_\top)\ \Delta\,\top : \textbf{Bool} \qquad (\textbf{Bool}_\bot)\ \Delta\,\bot : \textbf{Bool}$$

$$(\beta)\ \frac{(\beta : \alpha) \in \Delta}{\Delta\,\beta : \alpha} \quad (\text{Arrow})\ \frac{\Delta\,\tau_1 : t_1 \quad \Delta\,\tau_2 : t_2}{\Delta\,\tau_1 \mapsto \tau_2 : t_1 \to t_2}$$

For brevity, we omit rules $(\textbf{Int}_\delta)$, $(\textbf{Int}_\top)$ and $(\textbf{Int}_\bot)$.

Given $\forall \bar{\delta}, \bar{\beta}.C \Rightarrow \tau$ with underlying type $\forall \bar{\alpha}.t$, a shape environment $\Delta$, a sequence of instantiation (strictness) types $\tau_{ij}$ and a sequence of (underlying) types $t_i$ such that $\Delta\,\tau : t$ and for each $\Delta\,\beta_{ij} : \alpha_i$ it holds that $\Delta\,\tau_{ij} : t_i$.

*Example 4.* Coming back to the example above, we find $\Delta = \{\beta_{11} : \alpha_1, \beta_{12} : \alpha_1, \beta_{13} : \alpha_1, \beta_{21} : \alpha_2, \beta_{22} : \alpha_2, \beta_{23} : \alpha_2\}$ and $\tau_{11} = \delta_{f1} \mapsto \delta_{f2}$, $\tau_{12} = \delta_{g1} \mapsto \delta_{g2}$, $\tau_{13} = \delta_{e1} \mapsto \delta_{e2}$, $\tau_{21} = \delta_{f3} \mapsto \delta_{f4}$, $\tau_{22} = \delta_{g3} \mapsto \delta_{g4}$ and $\tau_{23} = \delta_1 \mapsto \delta_2$.

For each strictness constraint involving polymorphic variables, we need to generate the appropriate instantiated constraints. We observe that there are two possible kinds of polymorphic constraints:

$$\begin{array}{l} (1)\ \delta_1 \to \ldots \to \delta_n \to \beta \to \beta' \\ (2)\ \delta_1 \to \ldots \to \delta_n \to \beta \end{array}$$

where $n$ possibly equals zero. We have already seen constraints of the first kind. Constraints of the second kind state that given we have no information about $\delta_1, \ldots, \delta_n$ (which means $\delta_1, \ldots \delta_n = \top$), then we do not have any information about $\beta$ either. Such constraints prove to be useful if our analysis is not able to state any results.

*Example 5.* Consider an imported function $\mathsf{im} :: \forall \alpha.\alpha \mapsto \alpha \mapsto \alpha$ with no analysis information available. The best we can do is to assign $\mathsf{im}$ the following strictness type

$$\mathsf{im} : \forall \beta_1 \beta_2 \beta_3.\ \beta_3 \Rightarrow \beta_1 \mapsto \beta_2 \mapsto \beta_3$$

which states that we do not have any information about $\mathsf{im}$'s strictness behaviour.

We proceed by querying which relations hold and add in the appropriate constraints. The structure of the constraints allows us to exclude certain combination we have to test for. For example, given a constraint $\delta \to \beta \to \beta'$ we always know that variables $\delta$ and $\beta$ can only appear in a negative position and variable $\beta'$ in positive position in the type component.

We define *polarities* recursively: $\beta$ appears in *positive* position in $\tau'$ (written $\tau'[\beta^+]$) iff $\tau' = \beta$, or else $\tau' = \tau_1 \mapsto \tau_2$ and either $\tau_1[\beta^-]$ or $\tau_2[\beta^+]$. Similarly $\beta$ appears in *negative* position in $\tau'$ (written $\tau'[\beta^-]$) iff $\tau' = \tau_1 \mapsto \tau_2$ and $\tau_1[\beta^+]$ or $\tau_2[\beta^-]$.

We define the query relation for the first kind of constraints as follows

$$\mathcal{T}_{\alpha_i}(\Delta,\tau,\bar{\tau_i}) = \left\{ \delta_1 \to \cdots \to \delta_n \to [\![(\tau_{ij} \leq_s \tau_{ik})]\!] \;\middle|\; \begin{array}{l} \Delta\,\beta_{ij} : \alpha_i, \tau[\beta_{ij}^-], \\ \Delta\,\beta_{ik} : \alpha_i, \tau[\beta_{ik}^+], \\ \tau[\delta_1^-], \ldots, \tau[\delta_n^-] \\ C \models \delta_1 \to \cdots \to \delta_n \to \beta_{ij} \to \beta_{ik} \end{array} \right\}$$

In a similar style, we handle constraints of the second kind. We define

$$|\beta| = \beta \;\; |\delta| = \delta \;\; |\tau \mapsto \tau'| = |\tau'|$$

Then, the query relation for the second kind of constraints is as follows

$$\mathcal{T}'_{\alpha_i}(\Delta,\tau,\bar{\tau_i}) = \left\{ \delta_1 \to \cdots \to \delta_n \to |\tau_{ij}| \;\middle|\; \begin{array}{l} \Delta\,\beta_{ij} : \alpha_i, \tau[\beta_{ij}^+], \\ \tau[\delta_1^-], \ldots, \tau[\delta_n^-] \\ C \models \delta_1 \to \cdots \to \delta_n \to \beta_{ij} \end{array} \right\}$$

Note that we do not have to make the whole instance $\top$. For example, assume that we have $\delta \to \beta$ and $\beta$ is instantiated by $(\delta_1 \to \delta_2) \to (\delta_3 \to \delta_4)$. Then, it is sufficient to add in $\delta \to \delta_4$ only.

Both query relations can be implemented efficiently which ensures the effectiveness of analysing polymorphically typed programs.

$$\mathcal{T}(\Delta,\tau,\bar{\tau}) = \textstyle\bigwedge_{\alpha_i \in range(\Delta)} (\mathcal{T}_{\alpha_i}(\Delta,\tau,\bar{\tau_i}) \wedge \mathcal{T}'_{\alpha_i}(\Delta,\tau,\bar{\tau_i}))$$

Overall, the inference rule for polymorphic application is as follows

$$\text{(Var–Inst)} \quad \frac{\begin{array}{c} x : \forall\bar{\beta},\bar{\delta}.C \Rightarrow \tau \in \Gamma \quad t' = [\bar{t}/\bar{\alpha}]t \quad \Delta\,\tau : t \\ \text{for each } \Delta\,\beta_{ij} : \alpha_i \text{ generate fresh } \tau_{ij} \text{ such that } \Delta\,\tau_{ij} : t_i \\ C' = [\![[\bar{\delta}'/\bar{\delta}]C]\!] \wedge \mathcal{T}(\Delta,\tau,\bar{\tau}) \quad \tau' = [\bar{\tau}/\bar{\beta},\bar{\delta}'/\bar{\delta}]\tau \quad \bar{\delta}' \text{ new} \end{array}}{\Gamma, ((x :: \forall\bar{\alpha}.t)\sharp\bar{t}) :: t' \vdash_{inf} (C',\tau')}$$

where $(x :: \forall\bar{\alpha}.t)\sharp\bar{t}$ denotes instantiation of polymorphic variables $\bar{\alpha}$ by types $\bar{t}$.

We have left out the issue of building strictness types which are polymorphic in annotation and underlying types. It is straightforward to adjust the corresponding rules (Let), ($\forall$I) and ($\exists$I). We refer to previous work [4] for a detailed development.

### 4.2 Algebraic data types

We show how to handle non-recursive data types such as

$$\mathit{data}\,\mathsf{Maybe}\,\alpha = \mathsf{Just}\,\alpha \mid \mathsf{Error}\,\mathbf{Int}$$

which translates to the following strictness data type

$$\mathsf{Maybe}^{\delta_1}\,\beta\,\delta_2 = \mathsf{Just}\,\beta \mid \mathsf{Error}\,\delta_2$$

The top-most annotation $\delta_1$ tells us whether the data type is totally undefined (in case $\delta_1 = \bot$), or (in case $\delta_2 = \top$) the constructor may be known, but some of the arguments of the constructor are possibly undefined. The strictness variable $\beta$ represents the strictness behaviour of $\alpha$ and $\delta_2$ represents the strictness behaviour of $\mathsf{Error}$'s integer argument.

Value constructors are translated as follows:

$$\begin{array}{l} \mathsf{Just}\ : \forall \beta_0 \beta \delta_1 \delta_2.\ \beta_0 \to \beta \wedge \delta_1 \Rightarrow \beta_0 \mapsto \mathsf{Maybe}^{\delta_1}\ \beta\ \delta_2 \\ \mathsf{Error} : \forall \beta \delta_0 \delta_1 \delta_2.\ \delta_0 \to \delta_2 \wedge \delta_1 \Rightarrow \delta_0 \mapsto \mathsf{Maybe}^{\delta_1}\ \beta\ \delta_2 \end{array}$$

Note that in case of $\mathsf{Just}$ we do not constrain $\delta_2$. Similarly, we do not constrain $\beta$ in case of $\mathsf{Error}$.

In general, a non-recursive data type

$$\mathit{data}\,T\,\bar{\alpha} = k_1 \bar{t}_1 \mid \ldots \mid k_n \bar{t}_n$$

translates to

$$T^{\delta}\,\bar{\beta}\,\bar{\delta} = k_1 \bar{\tau}_{k_1} \mid \ldots \mid k_n \bar{\tau}_{k_n}$$

where each $\alpha_i$ is represented by $\beta_i$ and $\bar{\tau}_{k_i}$ results from $\Delta\,\bar{\tau}_{k_i} : \bar{t}_i$ where $\Delta = \{\beta_i : \alpha_i\}$ and $\bar{\delta}$ are the sequence of all freshly introduced annotation variables in $\bar{\tau}_{k_1}, \ldots, \bar{\tau}_{k_n}$.

Each value constructor $k_i : \forall \bar{\alpha}.t_{i1} \mapsto \ldots \mapsto t_{im} \mapsto T\,\bar{\alpha}$ translates to

$$\forall \bar{\beta}, \bar{\delta}.\ \delta \Rightarrow \tau_{i1} \mapsto \ldots \mapsto \tau_{im} \mapsto T^{\delta}\,\bar{\beta}\,\bar{\delta}$$

We find the following inference rule for case expressions:

$$(\text{Case})\quad \frac{\begin{array}{c} \Gamma, e :: T\,\bar{t}' \vdash_{\mathit{inf}} (C_0, T^{\delta}\,\bar{\tau}'\,\bar{\delta}') \quad \Delta\,\bar{\tau}'' : \bar{t}' \quad \bar{\delta}'' \text{ fresh} \\ \Gamma.\bar{x}_i : [\bar{\tau}''/\bar{\beta}, \bar{\delta}''/\bar{\delta}]\bar{\tau}_{k_i}, e_i : t \vdash_{\mathit{inf}} (C_i, \tau_i''') \quad \text{for } i = 1..n \\ \Delta\,\tau : t \\ C = C_0 \wedge \bigwedge_{i=1}^{n} C_i \wedge [\![(\bar{\tau}' \leq_s \bar{\tau}'')]\!] \wedge [\![(\bar{\delta}' \leq_s \bar{\delta}'')]\!] \wedge \delta \to (\bigwedge_{i=1}^{n} [\![(\tau_i''' \leq_s \tau)]\!]) \end{array}}{\Gamma, (\mathsf{case}\, e :: T\,\bar{t}' \,\mathsf{of}\, k_1 \bar{x}_1 \Rightarrow e_1 \mid \ldots \mid k_n \bar{x}_n \Rightarrow e_n) :: t \vdash_{\mathit{inf}} (C, \tau)}$$

For recursive algebraic data types we only consider the strictness properties of the top-most constructor. In effect, all information we have about recursive components is discarded. For example, the list description $[\beta]^{\delta}$ is interpreted as: $\delta$ describes the strictness behaviour of the topmost list constructor (so if it is $\bot$ then the whole list is $\bot$) while $\beta$ describes the strictness of the element in the first cons cell. So if a list-consuming function is strict in $\delta$ then it is safe to evaluate the list as far as its first constructor (i.e. to *weak head normal form*) before the function is applied. If the function is also strict in $\beta$ then the element in the head of the list can also be evaluated before the function is applied.

There have been many proposals for finding more accurate information for recursive data types, usually in the context of lists. For example *head-strictness* [13], *spine-strictness* (the shape of the expression is required) and *full-strictness* (all components of the expression are required) [12, 6]. We are currently investigating extensions to our analysis method to support these properties.

## 5 Implementation and empirical results

We have built a prototype implementation of the strictness analysis presented here for the Haskell compiler, GHC. The analysis engine is written in Haskell, while the Boolean constraints are handled by a specialized HORN constraint solver written in C, which represents Horn clauses as graphs with directed hyperedges. It supports the operations of: conjunction, existential quantification, copying and renaming. For polymorphic application we provide specialized support for efficiently determining (a minimal set of) conjunctions $(\delta_1 \wedge \cdots \wedge \delta_n)$ such that $(\delta_1 \wedge \cdots \wedge \delta_n) \rightarrow (\beta_1 \rightarrow \beta_2)$ or $(\delta_1 \wedge \cdots \wedge \delta_n) \rightarrow \beta$ based on finding paths from $\beta_1$ to $\beta_2$.

GHC's current strictness analysis [10] is based on the abstract interpretation approach of Burn *et al.* [1], made practical by a widening operation which eliminates all but weak-head normal form information about higher-order types and algebraic data types with more than a single constructor. The strictness analysis produces a strictness description for functional variables which is used by a later pass to optimise these functions safely. The description assigns each argument one of $S$ (for strict in weak head normal form, $L$ (for lazy) and $U(d_1, \ldots, d_n)$ for a strict argument of a *product* type (one with only a single constructor), where $d_i$ is the strictness description for argument $i$ of the constructor. For example the strictness descriptions for $+$ is $SS$ (for unboxed integers), snd is $U(LS)$ (strict in the pair and its second part) and f (from the introduction) is $LLLS$ (strict only in $b$). These strictness descriptions are written to an interface file for use in analysing modules which importing analysed functions.

This approach appears very weak, it loses all information about application of functional arguments. This would seem very unfortunate because even simple Haskell programs often have implicit higher-order arguments, because of type classes and their translation to dictionary passing. The simple function double x = x + x with type double :: Num a ⇒ a ↦ a becomes after processing, something like (for simplicity the Num dictionary is drastically simplified)

```
doubleCore :: (a ↦ a ↦ a, a ↦ a ↦ a) ↦ a ↦ a
doubleCore (plus,minus) x = plus x x
```

and the GHC strictness description discovered is $U(SL)L$ (strict in the plus part of the dictionary, but not in x). This apparent weakness is often avoided by aggressive inlining. Hence although we might expect doubleInt = (Int) double to be given an $L$ strictness description (no information), this only happens if inlining of double is switched off. Inlining of double plus dictionary specialization results in GHC's current analyser finding the desired $S$ strictness description.

Currently we do not write or read our strictness descriptions to/from the interface file, instead we use GHC's strictness info to build a (weak) Boolean strictness description for imported functions.

We can show that we are at least as accurate as GHC's current strictness analyser by mapping our Boolean strictness types to the simple sequences above and comparing the number of $S$ and $U$ annotations that occur. Note that this mapping throws away a considerable amount of the strictness information we have in our descriptions. We find all the strictness results that GHC finds. In addition, we often improve GHC's information.

Our experimental results are shown in Table 1, giving benchmark name (from the `nofib` suite) size (in no. of functions and lines), and original GHC compilation time. The data for each fixpoint rule is: the increase in compilation time when our strictness analysis phase is added, and the increase in number of U and S annotations found. Using (Fix) we are uniformly more accurate than GHC, but this is not the case for (FixC). An entry $x$-$y$ indicates $x$ U+S annotations found by (FixC) where GHC found L, and $y$ U+S annotations found by GHC where (FixC) found L. Two smaller `nofib` benchmark suites `imaginary` and `spectral` are shown in one summary line for space reasons.

| | | | GHC | Fix | | FixC | |
|---|---|---|---|---|---|---|---|
| Program | Functions | Lines | Comp (secs) | %+Time | Strict | %+Time | Strict |
| bspt | 271 | 2141 | 29.21 | 3473% | 71 | 1193% | 4-47 |
| cacheprof | 331 | 2151 | 37.64 | 662% | 4 | 111% | 4 |
| compress2 | 19 | 198 | 4.49 | 88% | — | 94% | 0-4 |
| ebnf2ps | 485 | 2927 | 34.78 | 1394% | 42 | 77% | 41-34 |
| fem | 103 | 1286 | 27.19 | 38% | 2 | 37% | 6-7 |
| fluid | 278 | 2401 | 34.74 | 124% | 22 | 52% | 18-50 |
| gamteb | 48 | 701 | 15.83 | 30% | 4 | 24% | 4-8 |
| gg | 292 | 812 | 19.81 | 89% | 9 | 40% | 12-20 |
| grep | 148 | 356 | 5.11 | 150% | 2 | 83% | 8-10 |
| hidden | 233 | 521 | 17.44 | 49% | 63 | 36% | 64-15 |
| hpg | 319 | 2067 | 15.32 | 2419% | 133 | 416% | 14-96 |
| infer | 398 | 584 | 13.33 | 237% | 12 | 84% | 14-80 |
| lift | 1943 | 2043 | 7.75 | 5% | 1 | 0% | 3-35 |
| maillist | 10 | 175 | 1.59 | 51% | 4 | 32% | 9-9 |
| mkhprog | 75 | 803 | 4.56 | 76% | 15 | 44% | 0-19 |
| pic | 96 | 527 | 13.26 | 30% | 25 | 23% | 5-11 |
| prolog | 164 | 539 | 7.01 | 227% | 22 | 60% | 14-31 |
| reptile | 205 | 1522 | 20.35 | 55% | 14 | 37% | 4-14 |
| rsa | 14 | 74 | 2.03 | 11% | 1 | 11% | 1 |
| symalg | 38 | 1146 | 16.79 | 212% | 2 | 8% | 5-8 |
| imaginary | 97 | 318 | 11.03 | 163% | 15 | 47% | 17-20 |
| spectral | 3385 | 21709 | 240.66 | 703% | 384 | 44% | 145-397 |

**Table 1.** Comparison with GHC built-in strictness analysis

The purpose of the comparison is mainly to show that the analysis is feasible, on a large and varied suite of programs. Our approach would already improve

GHC's current approach. We emphasize that the descriptions we are calculating carry much more strictness information.

Overall, we were surprised by the accuracy of the current analysis for GHC given its simplicity. Clearly the GHC implementors have ensured that the strictness analysis phase is well-integrated into the compiler as a whole and provides exactly the information they require. Of course in the comparison we are favoring GHC by only comparing on the information it collects. By writing and reading our strictness information to/from the interface file we should be able to improve our accuracy significantly. Better handling of recursive data types should also lead to much more accuracy. The (FixC) approach currently loses too much strictness information. Investigations so far suggest that (FixC) loses considerable information when the recursive call has an always constructed algebraic data type. Unfortunately since **Int**s are boxed this is very common. We are investigating how to avoid this information loss.

The analysis times are large, but not infeasible. At present the large analysis times seem to be caused by very large algebraic data types in combination with deeply nested bindings caused by GHC's aggressive inlining. At present the implementation is only a prototype and there are a number of obvious places for improvement: early elimination of variables during analysis; building a more efficient constraint solver; and, gracefully losing information when the analysis begins to take too much time. Another possibility worth investigating is a hybrid (Fix)/(FixC) approach which uses `FixC` after a certain number of fixpoint iterations. Arguably the (Fix) implementation could be used now in a highly optimizing compilation mode where we are willing to pay (considerable) time for the best possible performance.

## 6 Conclusion and Related Work

There is a huge amount of theoretical work of strictness analysis, but very little of it has translated into practice. Our aim with this work was to see how far we could go in producing an accurate strictness analyser for a real programming language while remaining practical.

The starting point for the work presented here is our binding time analysis using Boolean constraints [4]. The constraints that arise in strictness analysis are more complex (Horn formulae rather than simple implications) but the approaches are quite similar. Indeed, the two implementations share a considerable amount of code. We claim the analysis we have built is (almost) practical, and accurate compared to other implemented analysers.

Other implemented strictness analyses include Jensen *et al.* [5] which implements Burn *et al.*'s abstract interpretation approach using chaotic fixpoint iterations (which give a demand-driven evaluation of function descriptions that makes the higher-order function descriptions far more tractable). Unfortunately the chaotic fixpoint approach does not handle separate modular compilation, and the only experimental data they give is for very small programs. Seward [11] developed an abstract interpretation based strictness analysis in GHC which was capable of analysing large programs. The approach is essentially first-order, and loses precision for polymorphic functions. Results are given for small programs, and it is not clear how effective it is in practice.

The Clean functional language has a well regarded strictness analyser which finds strictness information through a process of abstract reduction [3]. It is not easy to compare our results with this system. The system is more complex than the system presented here and consequently difficult to show correct, and implement.

The theoretical approaches that are most similar to our approach are that of Kuo and Mishra [8], who gave the first constraint-based strictness analysis, and first used constraint-based fixpoints. They also gave an informal discussion of handling let-style polymorphism. Their approach is weak because it does not express conjunctive strictness information directly, but needs to do case by case analysis.

The closest work to ours is probably Jensen's framework for strictness [7]. He uses an inference system similar to ours to determine strictness types. Conditional properties are expressed using a ? operator, where $\phi?\psi$ is interpreted as $\perp$ if $\psi = \perp$ and $\phi$ otherwise. For example the strictness of f from the introduction is (in a format as close to ours as possible)

$$\mathsf{f} : \forall \delta_1\delta_2\delta_3\delta_4.\ (\delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4 \mapsto (\delta_2 \wedge \delta_3)?\delta_4) \wedge (\delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto \delta_4 \mapsto \delta_1?\delta_4)$$

Clearly the conditional expressions $\phi?\delta$ play the same role as $\delta$ in our (If) rule $\delta \to [\![(\tau_1 \leq_s \tau) \wedge (\tau_2 \leq_s \tau)]\!]$. However, the constraints in Jensen's approach are complicated and he needs to define meaning of new operators and rules for constraint simplification. For example, he reduces the constraint $\delta_1?\delta_2 \leq \delta_3 \wedge C$ to the (disjunctive) set of constraints $\{\delta_1 \leq \delta_3 \wedge C,\ \delta_2 = \perp \wedge C\}$ where each new constraint represents one possible way of proving the original constraint. This can be avoided by simply using well-understood Boolean constraints.

Jensen's work is based on ranked intersection types, which may appear to be much more expressive than the strictness types we consider. However, we conjecture that his approach captures no more strictness information At least for the examples in Jensen's paper the two approaches appear to give equivalent strictness information. There is no implementation of Jensen's approach that we are aware of, and the approach does not tackle let-style polymorphism and algebraic data structures.

We have presented an accurate and feasible strictness analysis based on HORN constraints, which we believe will eventually form the basis of a robust and practical strictness analysis in everyday use in a real compiler.

## Acknowledgments

We thank Harald Søndergaard and the referres for their valuable comments.

## References

1. Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. The theory and practice of strictness for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.
2. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft,

editor, *Proc. Second Int. Symp. Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1995.
3. E.Nocker. Strictness analysis using abstract reduction. In *Proceedings Conference on Functional Programming Languages and Computer Architectures (FPCA '93)*. ACM Press, 1993.
4. K. Glynn, P. J. Stuckey, M. Sulzmann, and H. Søndergaard. Boolean constraints for binding-time analysis. To appear in *Second Symposium on Programs as Data Objects (PADO-II)*.
5. K. Jensen, P. Hjæresen, and M. Rosendahl. Efficient strictness analysis for haskell. In *Proceedings of the First International Static Analysis Symposium*, volume 864 of *LNCS*, pages 346–362. Springer-Verlag, 1994.
6. T. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. on Programming Languages and Systems*, 19(5):751–803, 1997.
7. T. Jensen. Inference of polymorphic and conditional strictness properties. In *Proc. Twenty-Fifth ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM Press, 1998.
8. Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272, September 1989.
9. Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.
10. S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In Hammond and O'Donnell, editors, *Functional Programming, Glasgow 1993*, Springer Workshops in Computer Science, pages 201–220, 1993.
11. J. Seward. Beyond prototype implementations: Polymorphic projection analysis for glasgow haskell. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 382–399. Springer-Verlag, 1995.
12. P. Wadler. Strictness analysis on non-flat domains. In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood, 1987.
13. P. Wadler and J. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 385–407. Springer, 1987.