Strictness Meets Data Flow

Tom Schrijvers[1] and Alan Mycroft[2]

[1] Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be
[2] Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
http://www.cl.cam.ac.uk/users/am

Abstract. Properties of programs can be formulated using various techniques: dataflow analysis, abstract interpretation and type-like inference systems. This paper reconstructs strictness analysis (establishing when function parameters are evaluated in a lazy language) as a dataflow analysis by expressing the dataflow properties as an effect system. Strictness properties so expressed give a clearer operational understanding and enable a range of additional optimisations including *implicational strictness*. At first order strictness effects have the expected principality properties (best-property inference) and can be computed simply.

1 Introduction

Fosdick and Osterweil [3] introduced the notion of *path expressions* for data flow analysis. A path expression is a regular expression that summarises a program's control graph in terms of events of interest on program variables, branches, sequences and loops.

This paper adapts the idea of path expressions to strictness analysis for lazy functional languages such as Haskell [5]. In this setting, the events of interest are evaluations of (potentially) lazy values. What sets our approach apart from traditional forms of strictness analysis based on boolean functions [2, 7] or projections [9], is the combination with data flow information available in path expressions.

The combination of strictness and data flow information enables two additional forms of optimisation in addition to those based on conventional strictness and absence information. Firstly, it also captures *implicational strictness* between variables: whenever variable y is evaluated, then x has already been evaluated. Secondly, the path information reveals whether particular optimisations would apply to some but not all paths. Hence, it guides inlining to expose optimisation opportunities.

Lazy functional languages only evaluate expressions when required to progress the computation. This is similar to call-by-name in Algol60 or *normal order evaluation* in the lambda-calculus but with the additional 'laziness' requirement that repeated requests to evaluate the same expression only evaluate it once and make its value available immediately to subsequent requests. The standard implementation of a value is therefore a pointer to a *thunk*; multiple references to the same value become copies of this pointer. The thunk has two states: an unevaluated state (in which the *payload* is a pointer to code to compute the value and change the thunk's state) and an evaluated state in which the payload holds the value. GHC represents the is-evaluated flag by one of two code pointers; before evaluation the flag is the thunk (which does then not need storing in the payload, and which stores its result in payload offset zero), afterwards it is a simple "load payload offset zero" code sequence. Causing a thunk to move into evaluated state is called *forcing* it.

There are two costs borne by lazy languages which their eager counterparts do not pay. Firstly, a thunk which is created but inevitably later evaluated is pointless waste of resources. Classical strictness optimisation detects this at compile time, typically to create a pre-evaluated thunk when the expression-to-be-suspended appears and to optimise away the is-evaluated test at references to the value. (*Unboxing* optimisations remove the heap allocation too.) Secondly, the is-evaluated tests on thunks are repeated on repeated references to a value. For a single variable these can often be removed at control flow points which are dominated by a *force* operation, but a contribution of this work is that this can be generalised to consider dependencies between the evaluation state of two different variables—we call this 'implicational strictness'.

2 Type-and-Effect System

Source Language We consider a first-order functional language (see Figure 1), where a program p consists of a sequence of potentially recursive function definitions $f(x_1, \ldots, x_k) = e$. Expressions e are variables x, function application $f(e_1, \ldots, e_k)$, integer (natural number) literals n, constructor application $\texttt{succ}(e)$ and case elimination $\texttt{case}(e_1, e_2, x \to e_3)$ (where either e_2 is returned if e_1 evaluates to 0, or e_3 is returned if e_1 evaluates to $\texttt{succ}(x)$).

Types and Effects Figure 1 lists the syntax for types and effects. Value types τ consist so far only of the type `Nat` of naturals;[3] function types are of the form $\tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$ where τ_i are the argument types, τ the return type, and ϕ its effect.

An effect ϕ is either a *parameter* x_i denoting the effect of evaluating the ith function argument x_i (variables bound by `case` are effectively eager), the constant 0 for non-terminating programs, the constant 1 for effect-free programs, the sequential composition of effects $\phi_1 \cdot \phi_2$ and non-deterministic choice of effects $\phi_1 + \phi_2$. By abuse of notation, a name x_i denotes both a source-level variable and its associated effect.

[3] This means that, not counting effects, all variables and functions have exactly one type, so we do not need to introduce polymorphic types to discuss the principality of inference for types containing effects.

Source Language

$$\begin{array}{lll}
\text{programs} \; p & ::= & d_1 \;\cdots\; d_m \\
\text{definitions} \; d & ::= & f(x_1, \ldots, x_k) = e \\
\text{expressions} \; e & ::= & x \\
 & \mid & f(e_1, \ldots, e_k) \\
 & \mid & n \\
 & \mid & \texttt{succ}(e) \\
 & \mid & \texttt{case}(e_1, e_2, x \to e_3)
\end{array}$$

Types and Effects

$$\begin{array}{lll}
\text{effects} & \phi, \psi ::= & x_i \\
 & \mid & 0 \\
 & \mid & 1 \\
 & \mid & \phi_1 \cdot \phi_2 \\
 & \mid & \phi_1 + \phi_2 \\
\text{value types} & \tau \; ::= & \texttt{Nat} \\
\text{function types} & \sigma \; ::= & \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau
\end{array}$$

$\phi_1 + \phi_2 \equiv \phi_2 + \phi_1$	(1)	$\phi \cdot 0 \equiv 0$	(7)
$(\phi_1 + \phi_2) + \phi_3 \equiv \phi_1 + (\phi_2 + \phi_3)$	(2)	$\phi \cdot 1 \equiv \phi$	(8)
$(\phi_1 \cdot \phi_2) \cdot \phi_3 \equiv \phi_1 \cdot (\phi_2 \cdot \phi_3)$	(3)	$1 \cdot \phi \equiv \phi$	(9)
$\phi + \phi \equiv \phi$	(4)	$\phi_3 \cdot (\phi_1 + \phi_2) \equiv \phi_3 \cdot \phi_1 + \phi_3 \cdot \phi_2$	(10)
$\phi + 0 \equiv \phi$	(5)	$(\phi_1 + \phi_2) \cdot \phi_3 \equiv \phi_1 \cdot \phi_3 + \phi_2 \cdot \phi_3$	(11)
$0 \cdot \phi \equiv 0$	(6)	$x \cdot \phi \cdot x \equiv x \cdot \phi$	(12)

Fig. 1. Syntax and Equivalence Laws

Operational Semantics Figure 2 lists the small-step operational semantics of our language, inspired by Launchbury's big-step semantics [6] for lazy evaluation. The judgement $\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'$ denotes a small step from expression e and environment ρ to expression e' and environment ρ'. Values n do not reduce. An environment ρ is a map from variables to unevaluated expressions (denoted $x \mapsto e$) or evaluated values (denoted $x = n$). An expression e is an *evaluated* natural number n in ρ, denoted $e \overset{\rho}{=} n$, iff e is n, or e is a variable x and $(x = n) \in \rho$; otherwise e is *unevaluated* in ρ, denoted $e \overset{\rho}{\neq}$.

Rule (Var1) evaluates one step of unevaluated variable x, while rule (Var2) recognises that x has been fully evaluated and issues effect x. Subsequent occurrences of x are handled by rule (Var3). Rule (App) for function call is noteworthy: it replaces the call by the function body, and updates the environment with mappings from the formal arguments to the actual arguments. Following Launchbury, we assume that a renamed-apart copy of the function definition (including internal `case` bindings) is used to avoid name capture.

Not listed in the figure is the usual context rule, with context $\mathbb{C} ::= \texttt{succ}(\cdot) \mid \texttt{case}(\cdot, e_2, x \to e_3)$:

$$(\textsc{Context}) \; \frac{\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'}{\rho; \mathbb{C}[e] \overset{\phi}{\rightarrowtail} \rho'; \mathbb{C}[e']}$$

The trace ϕ of a single step is either 1 or an x (for some variable x). The transitive closure judgement $\rho; e \overset{\phi}{\rightarrowtail}_* \rho'; e'$ sequences the effects $\phi_1, \ldots, \phi_n$ of its

$$\boxed{\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'}$$

$$(\textsc{Var1})\ \frac{\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'}{\rho[x \mapsto e]; x \overset{\phi}{\rightarrowtail} \rho'[x \mapsto e']; x} \qquad (\textsc{Var2})\ \frac{}{\rho[x \mapsto n]; x \overset{x}{\rightarrowtail} \rho[x = n]; n}$$

$$(\textsc{Var3})\ \frac{}{\rho; x \overset{1}{\rightarrowtail} \rho; n}\ \textit{if } (x = n) \in \rho \qquad (\textsc{Succ})\ \frac{}{\rho; \texttt{succ}(n) \overset{1}{\rightarrowtail} \rho; n+1}$$

$$(\textsc{App})\ \frac{}{\rho; f(e_1, \ldots, e_k) \overset{1}{\rightarrowtail} \rho[x_1 \mapsto e_1, \ldots, x_k \mapsto e_k]; e}\ \textit{if } f(x_1, \ldots, x_k) = e$$

$$(\textsc{Case2})\ \frac{}{\rho; \texttt{case}(0, e_2, x \rightarrow e_3) \overset{1}{\rightarrowtail} \rho; e_2}$$

$$(\textsc{Case3})\ \frac{}{\rho; \texttt{case}(n+1, e_2, x \rightarrow e_3) \overset{1}{\rightarrowtail} \rho[x = n]; e_3}$$

Fig. 2. Small-Step Operational Semantics

constituent steps into a string $\phi_1 \cdot \ldots \cdot \phi_n$. Note that 0 and + effects only arise in the type-and-effect system, not through evaluation. Because the semantics does not *garbage collect* the local variables introduced by rule (App), we write $\phi|_{\{x_1,\ldots,x_n\}}$ to project ϕ onto a set of variables of interest $\{x_1, \ldots, x_n\}$. This is needed to express effect system soundness (Section 2.4).

Effect Algebra In addition to syntactic equivalence, equivalence of effects is governed by a number of laws, listed in Figure 1. These are the forms and equality laws for regular languages (operators + and $\cdot$ with units 0 and 1 and with $\cdot$ distributing over +) over alphabet $\{x_1, \ldots, x_k\}$, but with the additional equation (12) expressing the fact that repeated elements later (but not earlier) in a sequence are redundant.

This last law is motivated by the meaning of the parameters: x_i denotes that x_i is evaluated *at the latest* at this point. Once the effect has taken place, x_i is definitely in evaluated form. The conservative approximation lies in the fact whether x_i is evaluated at this point, or has already been evaluated before. Hence, in $x_i \cdot x_i$ we know that x_i is evaluated at the latest at the first occurrence of x_i. The second occurrence is thus redundant, because we know that x_i is already evaluated before it. In summary, we conclude that $x_i \cdot x_i \equiv x_i$.

Definition 1 (Disjunctive Normal Form). *The Disjunctive Normal Form ϕ_n of any effect ϕ is the effect obtained after exhaustive rewriting with the AC rewrite system comprised of the equivalence laws (4)–(12) interpreted as left-to-right rewrite rules. We also denote the DNF of ϕ as dnf(ϕ).*

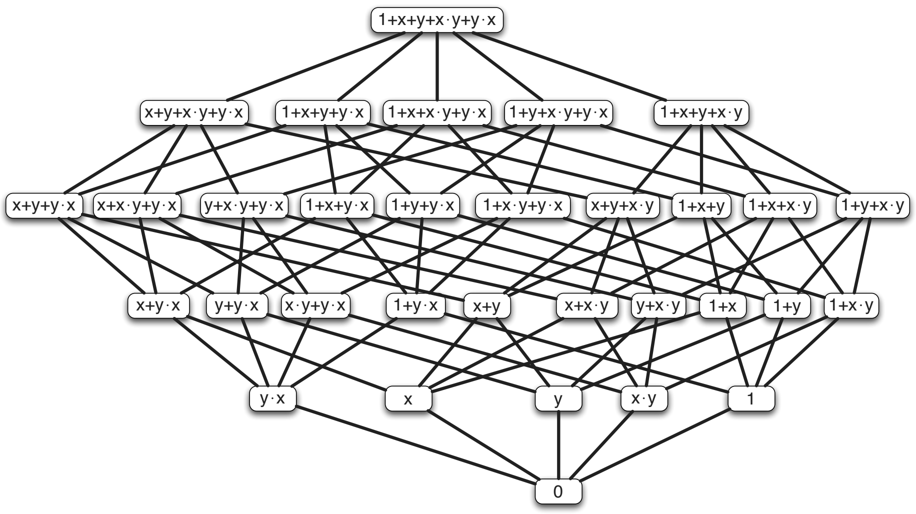

Fig. 3. The 32 different effects involving the two variables x and y

The Disjunctive Normal Form (DNF) is a non-deterministic choice of sequential compositions. Each effect has a DNF that is unique modulo associativity and commutativity. All equivalent effects have the same DNF.

Number of Distinct Effects The number of distinct effects over a finite set of parameters is finite. For instance, the set of different effects over two parameters contains 32 elements (see Figure 3). The lines in the figure denote the "subeffect" relation, which is explained later.

The basic building blocks for effects are all permutations of k variables with $0 \leq k \leq n$; there are $\sum_{k=0}^{n} \frac{n!}{k!}$ such building blocks for n variables. Note that the permutation of length 0 denotes the effect 1. For instance, for $n = 1$ there are 2 building blocks: 1 and x_1. For $n = 2$ there are 5: 1, x_1, x_2, $x_1 \cdot x_2$ and $x_2 \cdot x_1$.

These building blocks are combined into effects with the + operator; this yields $2^{\sum_{k=0}^{n} \frac{n!}{k!}}$ distinct effect terms that range over n parameters. Note that if none of the building blocks is selected, we obtain the effect 0. For instance, for $n = 1$ there are 4 distinct effects, and for $n = 2$ there are 32 distinct effects.

Definition 2 (Chaos). *We define the chaos effect X^* ranging over a set of parameters $X = \{x_1, \ldots, x_n\}$ as*

$$X^* = \underbrace{(1 + x_1 + \ldots x_n) \cdot \ldots \cdot (1 + x_1 + \ldots x_n)}_{n\ times}$$

Bitvector Representation The observation about the composition of effects from building blocks suggests a bitvector representation $\bar{b}$ for effects where bit b_i denotes whether the ith building block is present or not. The ordering of building

x_1	1	ϕ
0	0	0
0	1	1
1	0	x_1
1	1	$x_1 + 1$

$$\begin{tikzcd}
 & x_1 + 1 \arrow[dl, dash] \arrow[dr, dash] & \\
1 \arrow[dr, dash] & & x_1 \arrow[dl, dash] \\
 & 0 &
\end{tikzcd}$$

Fig. 4. The effect domain ranging over a single effect variable x_1.

blocks in the bitvector representation may be chosen arbitrarily. Figure 4 lists the distinct effects for $n = 1$ with their bitvector representation.

Subeffects Effects have a natural (partial) ordering—the subeffect ordering.

Definition 3. *The subeffecting relation $<:$ is the minimal relation that satisfies (up to $\equiv$) the following axiom:*

$$\phi_1 <: \phi_1 + \phi_2$$

We say that ϕ_1 is a subeffect of $\phi_1 + \phi_2$.

Note that this relation is indeed a partial order. For instance, the reflexivity property $\phi_1 <: \phi_1$ follows from choosing $\phi_2 \equiv 0$. The minimal element is 0 and the maximal element is chaos X^*. The subeffect lattice for a single variable x_1 is shown in Figure 4. The least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operators on this lattice are defined in the usual manner. Observe that they correspond to bitwise *or* $\vee$ and bitwise *and* $\wedge$ on the bit vector representation.

The $<:$ relation and $\sqcap$ and $\sqcup$ operations are lifted pointwise to function types:

$$\begin{aligned}
\bar{\tau} \xrightarrow{\phi_1} \tau <: \bar{\tau} \xrightarrow{\phi_2} \tau \quad &\text{iff} \quad \phi_1 <: \phi_2 \\
(\bar{\tau} \xrightarrow{\phi_1} \tau) \sqcap (\bar{\tau} \xrightarrow{\phi_2} \tau) \quad &= \quad \bar{\tau} \xrightarrow{\phi_1 \sqcap \phi_2} \tau
\end{aligned}$$

and (later) to environments Γ.

2.1 Type-and-Effect Inference System

The expression typing judgement is of the form $\Gamma \vdash e : \tau \,\&\, \phi$, and denotes that expression e has type τ and its evaluation has effect ϕ with respect to the type environment Γ. In the first-order language there are separate syntactic variable names for values (x) and functions (f). Type assumptions Γ contain constraints such as $x : \tau \,\&\, \phi$ and $f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$.

Figure 5 lists the rules for the type-and-effect system. Rule (Var) looks up the type of a function argument in the type environment and returns the effect corresponding to that argument. Rule (App) makes sure that the types of the arguments match the function typing in the environment.

Rules (Lit) and (Succ) cover the predefined constants. Note that to model standard implementation of arithmetic, the `succ` data constructor is strict in its argument e: the effect of evaluating `succ`(e) is the effect of evaluating e.

$$\boxed{\Gamma \vdash e : \tau \,\&\, \phi} \qquad (\textsc{Var})\ \frac{}{\Gamma \vdash x : \tau \,\&\, \phi}\ \mathit{if}\ (x : \tau \,\&\, \phi) \in \Gamma$$

$$(\textsc{Lit})\ \frac{}{\Gamma \vdash n : \texttt{Nat} \,\&\, 1} \qquad (\textsc{Succ})\ \frac{\Gamma \vdash e : \texttt{Nat} \,\&\, \phi}{\Gamma \vdash \texttt{succ}(e) : \texttt{Nat} \,\&\, \phi}$$

$$(\textsc{App})\ \frac{\Gamma \vdash e_i : \tau_i \,\&\, \phi_i \quad (i \in 1..k)}{\Gamma \vdash f(e_1, \ldots, e_k) : \tau \,\&\, \phi[\overline{\phi_i / x_i}]}\ \mathit{if}\ (f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau) \in \Gamma$$

$$(\textsc{Case})\ \frac{\Gamma \vdash e_1 : \texttt{Nat} \,\&\, \phi_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, \phi_2 \qquad \Gamma[x : \texttt{Nat} \,\&\, 1] \vdash e_3 : \tau \,\&\, \phi_3}{\Gamma \vdash \texttt{case}(e_1, e_2, x \to e_3) : \tau \,\&\, \phi_1 \cdot (\phi_2 + \phi_3)}$$

$$\boxed{\Gamma \vdash f(\bar{x}) = e} \qquad (\textsc{Def})\ \frac{\Gamma[\bar{x} : \bar{\tau} \,\&\, \bar{x}] \vdash e : \tau \,\&\, \phi}{\Gamma \vdash f(\bar{x}) = e}\ \mathit{if}\ (f : \bar{\tau} \xrightarrow{\phi} \tau) \in \Gamma$$

$$\boxed{\Gamma \vdash \bar{d}} \qquad (\textsc{Prog})\ \frac{\Gamma \vdash d_1 \quad \cdots \quad \Gamma \vdash d_n}{\Gamma \vdash d_1\ \cdots\ d_n}$$

Fig. 5. Type-and-Effect Inference Rules

A function definition $f(\bar{x}) = e$ is well-typed w.r.t. environment Γ, denoted $\Gamma \vdash f(\bar{x}) = e$, if the function's typing is recorded in the environment and the function's body is well-typed w.r.t. that typing (Rule (Def)). A program $\bar{d}$ is well-typed w.r.t. environment Γ, denoted $\Gamma \vdash \bar{d}$, if all its definitions are well-typed (Rule (Prog)).

2.2 Principality

Theorem 1 (Unique Non-Recursive Function Typing). *For any Γ, there is* at most *one typing $f : \bar{\tau} \xrightarrow{\phi} \tau$ such that $\Gamma, f : \bar{\tau} \xrightarrow{\phi} \tau \vdash f(\bar{x}) = e$, if f is not recursive, i.e., e does not contain a call $f(\bar{e})$.*

Note that due to our restricted setting with only one type Nat there is in fact *exactly* one such function typing.

Recursive functions admit multiple typings that differ in their effect. For instance, $f(x_1) = f(x_1)$ admits typings $f : \texttt{Nat} \xrightarrow{\phi} \texttt{Nat}$ for any effect ϕ. Similarly, $f(x_1, x_2) = \texttt{case}(x_1, x_2, y \to f(y, x_2))$ has well-typings $x_1 \cdot (x_2 + \phi)$ for any ϕ. The cause of these multiple typings is the (Def) rule, which defines a recursive function's well-typing in terms of itself, i.e., as a fixpoint. Any fixpoint is a valid solution. This issue of self-reference also exists in traditional dataflow analysis. Usually, in that context, the analysis domain naturally has a lattice structure and the least (sometimes greatest) fixpoint in that lattice is the preferred one. We follow the same approach.

If two different well-typings are possible, then their greatest lower bound is also a well-typing.

Lemma 1. *For all environments Γ_1, Γ_2 and programs $\bar{d}$, if $\Gamma_1 \vdash \bar{d}$ and $\Gamma_2 \vdash \bar{d}$, then $\Gamma_1 \sqcap \Gamma_2 \vdash \bar{d}$.*

As the lattice is finite, it follows that there is a unique minimal well-typing: the principal type.

Corollary 1 (Principality). *For all environments Γ_1, Γ_2 and programs $\bar{d}$, if $\Gamma_1 \vdash \bar{d}$ and $\Gamma_2 \vdash \bar{d}$, then there exists an environment Γ such that $\Gamma <: \Gamma_1$ and $\Gamma <: \Gamma_2$ and $\Gamma \vdash \bar{d}$.*

In contrast to the data-flow-analysis approach that we follow, effect systems typically have a coercion rule:

$$(\textsc{Coerce})\ \frac{\Gamma, e : \tau_1 \,\&\, \phi_1}{\Gamma \vdash e : \tau_2 \,\&\, \phi_2}\ \textit{if } \tau_1 <: \tau_2 \textit{ and } \phi_1 <: \phi_2$$

but this is unnecessary here because (i) effects can express non-deterministic choice using $+$, and (ii) in the first-order setting, subeffecting only applies covariantly and thus all coercions in a judgement can be pushed to the root of the proof tree and thus merged into the $<:$ of principality.

2.3 Connection to Traditional Type Inference

The type-and-effect system we have defined has the property that type inference can be done first, followed by effect inference. Type, or type-and-effect, inference can be explained in terms of reconstructing information removed by erasure operators. Erasure of types, and reconstructing types without effects is standard. So we now consider an erasure operator which removes effects from expression types-and-effects and from function types yielding *traditional types* (which in our case are *simple types* but could equally be Hindley-Milner polymorphic types), and state various results. Effect erasure is defined as follows:

$$\epsilon(\tau \,\&\, \phi) = \epsilon(\tau) \qquad \epsilon(\texttt{Nat}) = \texttt{Nat} \qquad \epsilon(\bar{\tau} \xrightarrow{\phi} \tau) = \bar{\tau} \to \tau$$

and lifted to environments Γ as usual.

Results A well-typing ($\vdash$) in the type-and-effect system is also a traditional well-typing ($\vdash_T$). Conversely, a well-traditional-typing always has a well-typing in the type-and-effect system (i.e. the type-and-effect system is a conservative extension).

Theorem 2 (Conservative Extension).

$$(\forall e, \Gamma, \tau) \quad \epsilon(\Gamma) \vdash_T e : \epsilon(\tau) \;\Leftrightarrow\; (\exists \phi) \;\; \Gamma \vdash e : \tau \,\&\, \phi$$

2.4 Effect System Soundness

We have presented a semantics and an effect system for our simple language and now address their consistency. To establish soundness, we show an enriched form of progress and preservation lemmas,[4] after an auxiliary definition.

Definition 4 (Compatible Environments). *A typing environment Γ is compatible with an evaluation environment ρ iff $\Gamma = tenv(\rho)$ where*

$$tenv(\rho) = \{x : \texttt{Nat} \,\&\, x \mid (x \mapsto e) \in \rho\} \cup \{x : \texttt{Nat} \,\&\, 1 \mid (x = n) \in \rho\}$$

Progress expresses that a well-typed non-value expression is never stuck.

Lemma 2 (Progress).

$$(\forall \rho_1, e_1, \tau, \phi_1) \quad tenv(\rho_1) \vdash e_1 : \tau \,\&\, \phi_1 \quad \Rightarrow \quad (\exists \rho_2, e_2, \phi_2) \quad \rho_1; e_1 \overset{\phi_2}{\rightarrowtail} \rho_2; e_2$$

where e_1 is not a value.

The preservation lemma expresses that the types and effects before and after an evaluation step are related appropriately: the type is the same and the original effect subsumes the concatenation of the evaluation trace and the new effect.

Lemma 3 (Preservation).

$$\begin{aligned}(\forall \rho_1, \rho_2, e_1, e_2, \tau, \phi_1, \phi_2) \quad & tenv(\rho_1) \vdash e_1 : \tau \,\&\, \phi_1 \quad \wedge \quad \rho_1; e_1 \overset{\phi_2}{\rightarrowtail} \rho_2; e_2 \\ \Rightarrow \quad (\exists \phi_3) \quad & tenv(\rho_2) \vdash e_2 : \tau \,\&\, \phi_3 \quad \wedge \quad (\phi_2 \cdot \phi_3)|_{dom(\rho_1)} <: \phi_1\end{aligned}$$

3 Inference Algorithm

Figure 6 lists our first-order inference algorithm. The inference judgement for expressions is of the form $\Gamma \overset{\mathcal{A}}{\vdash} e : \tau \,\&\, \phi \mid C$, which denotes that type τ and effect ϕ are inferred for expression e with respect to environment Γ and with Γ and τ subject to a set C of type equality constraints of the form $\tau = \tau'$.[5]

The type-and-effect information in the inference algorithm are essentially independent. The type-related part of the algorithm corresponds to traditional type inference as discussed earlier.

The effect inference for expressions is fairly straightforward. A composite expression's effect is a composite effect, composed from the components' effects. Note that in each case the minimal effect of an expression is returned.

The hardest part of effect inference takes place for a function definition. For recursive calls during the inference of the function body, we use a meta-effect γ as a place-holder. The body's inference returns an effect ϕ for the function that potentially mentions γ. In order to obtain a proper effect for the function, the

[4] The traditional lemmas are recovered through effect erasure.

[5] $\models C$ denotes that C is satisfiable, usually established by unification.

$$\boxed{\Gamma \vdash^{\mathcal{A}} e : \tau \,\&\, \phi \mid C}$$

$$(\textsc{Var})\ \frac{(x : \tau \,\&\, \phi) \in \Gamma}{\Gamma \vdash^{\mathcal{A}} x : \tau \,\&\, \phi \mid \mathit{true}} \qquad (\textsc{Lit})\ \frac{}{\Gamma \vdash^{\mathcal{A}} n : \mathtt{Nat} \,\&\, 1 \mid \mathit{true}}$$

$$(\textsc{Succ})\ \frac{\Gamma \vdash^{\mathcal{A}} e : \tau \,\&\, \phi \mid C}{\Gamma \vdash^{\mathcal{A}} \mathtt{succ}(e) : \mathtt{Nat} \,\&\, \phi \mid C \wedge \tau = \mathtt{Nat}}$$

$$(\textsc{App})\ \frac{f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau \in \Gamma \qquad \Gamma \vdash^{\mathcal{A}} e_i : \tau_i' \,\&\, \phi_i \quad (i \in 1..k) \mid C_i}{\Gamma \vdash^{\mathcal{A}} f(e_1, \ldots, e_k) : \tau \,\&\, \phi[\overline{\phi_i}/\overline{x_i}] \mid \bar{C} \wedge \bar{\tau} = \bar{\tau}'}$$

$$(\textsc{Case})\ \frac{\begin{array}{c}\Gamma \vdash^{\mathcal{A}} e_1 : \tau_1 \,\&\, \phi_1 \mid C_1 \\ \Gamma \vdash^{\mathcal{A}} e_2 : \tau_2 \,\&\, \phi_2 \mid C_2 \qquad \Gamma[x : \mathtt{Nat} \,\&\, 1] \vdash^{\mathcal{A}} e_3 : \tau_3 \,\&\, \phi_3 \mid C_3\end{array}}{\Gamma \vdash^{\mathcal{A}} \mathtt{case}(e_1, e_2, x \to e_3) : \tau_2 \,\&\, \phi_1 \cdot (\phi_2 + \phi_3) \mid C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 = \mathtt{Nat} \wedge \tau_2 = \tau_3}$$

$$\boxed{\Gamma_1 \vdash^{\mathcal{A}} f(\bar{x}) = e : \Gamma_2}$$

$$(\textsc{Def})\ \frac{\begin{array}{c}\Gamma[f : \overline{\mathtt{Nat}} \xrightarrow{\gamma} \mathtt{Nat}, \bar{x} : \overline{\mathtt{Nat}} \,\&\, \bar{x}] \vdash^{\mathcal{A}} e : \tau \,\&\, \phi \mid C \\ \models C \wedge \tau = \mathtt{Nat}\end{array}}{\Gamma \vdash^{\mathcal{A}} f(\bar{x}) = e : \Gamma, f : \overline{\mathtt{Nat}} \xrightarrow{\psi} \mathtt{Nat}}\ \mathit{if}\ \psi = \mathit{lfp}(\lambda\gamma.\phi)$$

$$\boxed{\vdash^{\mathcal{A}} \bar{d} : \Gamma}$$

$$(\textsc{Prog})\ \frac{\emptyset \vdash^{\mathcal{A}} d_1 : \Gamma_1 \qquad \cdots \qquad \Gamma_{n-1} \vdash^{\mathcal{A}} d_n : \Gamma_n}{\vdash^{\mathcal{A}} d_1 \ \cdots\ d_n : \Gamma_n}$$

Fig. 6. Syntax-Directed Inference Algorithm

equation $\phi <: \gamma$ must be solved. The least solutions of this inequation is obtained as the least fixpoint of $\mu\gamma.\phi$, starting from 0. The number of iterations needed to obtain the least fixpoint is bounded from above by the number of distinct variable permutations, but may be much smaller in practice.

Example 1. Consider the function definition $g(x_1, x_2) = \mathtt{case}(x_1, x_2, y \to g(y, x_2))$ with effect equation $x_1 \cdot (x_2 + \phi[1/x_1, x_2/x_2]) <: \phi$. We obtain the least fixpoint in two steps, and confirm it in the third step:

$$\begin{aligned}&\phi_0 \equiv 0 \\ &\phi_1 \equiv x_1 \cdot (x_2 + \phi_0[1/x_1, x_2/x_2]) \equiv x_1 \cdot x_2 \\ &\phi_2 \equiv x_1 \cdot (x_2 + \phi_1[1/x_1, x_2/x_2]) \equiv x_1 \cdot x_2\end{aligned}$$

3.1 Properties

Theorem 3 (Soundness & Completeness wrt. the Inference System). *If $\vdash^{\mathcal{A}} \bar{d} : \Gamma$, then $\Gamma \vdash \bar{d}$, for any program $\bar{d}$ and environment Γ. If $\Gamma \vdash \bar{d}$, then $\vdash^{\mathcal{A}} \bar{d} : \Gamma'$, for any program $\bar{d}$ and environment Γ and for some Γ'.*

Theorem 4 (Principality). *If $\Gamma \vdash \bar{d}$ and $\vdash^{\mathcal{A}} \bar{d} : \Gamma'$, then $\Gamma' <: \Gamma$ for any program $\bar{d}$ and environments Γ, Γ'.*

Theorem 5 (Termination). *The inference algorithm terminates for any program $\bar{d}$.*

4 Optimisations

A number of different optimisations are possible.

4.1 Standard Strictness Analysis and Optimisations

Our strictness domain is more expressive than the Boolean expressions used in traditional strictness analysis. The abstraction relation α maps our effects to Boolean expressions.

$$\begin{array}{lll} \alpha(1) = 1 & \alpha(\phi_1 \cdot \phi_2) = \alpha(\phi_1) \wedge \alpha(\phi_2) & \alpha(x_i) = x_i \\ \alpha(0) = 0 & \alpha(\phi_1 + \phi_2) = \alpha(\phi_1) \vee \alpha(\phi_2) & \end{array}$$

Moreover $\alpha(\phi[\phi'/x]) = \alpha(\phi)[\alpha(\phi')/x]$.

Lemma 4 (Well-definedness). *Equivalent effects abstract to equivalent boolean functions:*

$$(\forall \phi_1, \phi_2) \quad \phi_1 \equiv \phi_2 \quad \Rightarrow \quad \alpha(\phi_1) \equiv \alpha(\phi_2)$$

The converse does not hold. Consider $\phi_1 = 1 + x_1$ and $\phi_2 = 1$. While $\phi_1 \not\equiv \phi_2$, we do have that $\alpha(\phi_1) \equiv \alpha(\phi_2) \equiv 1$. Hence, the α mapping is an abstraction because it loses information.

Theorem 6 (Complete abstraction). *Given program $\bar{d}$ and writing $\vdash^{\mathcal{S}}$ for standard boolean strictness inference using boolean functions, then $\vdash^{\mathcal{S}} \bar{d} : \alpha(\Gamma) \Leftrightarrow \vdash^{\mathcal{A}} \bar{d} : \Gamma$.*

The following two optimisations are enabled by standard strictness analysis.

Eager Evaluation If a function is strict in an argument, then that argument may be evaluated before the function call. A function is strict in argument x_i if $\alpha(\phi[0/x_i]) \equiv 0$. Since 0 is the only effect ϕ for which $\alpha(\phi) = 0$, we can equally check argument strictness by testing $\phi[0/x_i] \equiv 0$.

Loop Detection As in traditional strictness, if an expression e has effect 0, then its evaluation does not terminate. Hence, it may be replaced by `loop()`:
- If `loop` is defined as `loop() = loop()`, the transformed code should run in constant space, whereas e may not.
- Alternatively, defining `loop() = error("loop!")`, using a Haskell feature, transforms the code to abort evaluation and report non-termination to the programmer.

4.2 Inlining to expose Standard Strictness Optimisation

If a function is not strict in an argument, standard strictness optimisations do not apply. However, not being strict in an argument may mean either that the function never evaluates its argument or only sometimes evaluates it. In the latter case, there are one or more branches that do not evaluate the argument and one or more that do evaluate it. Inlining and floating the actual arguments into the branches, may effectively enable standard strictness optimisations. Our effects can be useful for guiding inlining.

For instance, if $f(x_1) = e$ has effect 1, this means that inlining of f will not uncover any opportunities for strictness optimisation, while $1 + x_1$ promises opportunities for parameter x_1.

Example 2. The function $f(x_1, x_2) = \texttt{case}(x_1, 0, y \to x_2)$ has type

$$f : \texttt{Nat}, \texttt{Nat} \xrightarrow{x_1 \cdot (1+x_2)} \texttt{Nat}$$

which provides no direct opportunity for strictness optimisation of x_2. However, after inlining, strictness optimisation can be applied to the second branch of f.

Note that in general inlining of a single function f may not be sufficient to uncover optimisation opportunities. Take f to be defined as $f(x_1) = g(x_1)$ where g has the effect $1 + x_1$ to illustrate this point. In the worse case, we may need to inline successively all the functions in the program to expose a strictness optimisation opportunity guaranteed by the typing.

4.3 Absent Argument Optimisation

If a function does not use (i.e. evaluate) its argument, then the argument is effectively dead code. So instead of the actual argument, the caller may provide a dummy argument or even no argument at all, i.e. an absent argument [9].

A function of type $f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$ does not evaluate its ith argument (on any path which can return) if $x_i \notin \phi$. For instance, a function of type $f : \texttt{Nat} \xrightarrow{1} \texttt{Nat}$ does not evaluate its argument. Hence, the function definition can be rewritten from $f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots) = e$ to $f(\ldots, x_{i-1}, x_{i+1}) = e$, and likewise the ith argument may be dropped from all calls in the program. It is important to do Loop Detection Optimisation first (which replaces paths, including possible references to x_i on them, which can never return with `loop()`), consider e.g. $f(x) = \texttt{case}(x, f(x), y \to f(x))$.

Note that absent argument information is not present in the traditional strictness domain. There we have that $\alpha(1) = 1 = \alpha(1 + x_1)$.

4.4 Implicational Strictness

A standard optimisation exploits the explicit *intraprocedural* data flow and avoids consecutive evaluation of the same variable. For instance, the second occurrence of x in $\texttt{case}(x, \texttt{case}(x, e_1, y \to e_3), z \to e_4)$, is known to have been evaluated already. So the expression can be replaced with $\texttt{case}(x, \texttt{case\#}(x, e_1, y \to$

$e_3), z \to e_4)$, where `case#` does not force its argument (i.e. reads the payload of its discriminant directly):

$$(\textsc{Case\#})\ \frac{}{\rho; \texttt{case\#}(e_1, e_2, x \to e_3) \overset{1}{\rightarrowtail} \rho; \texttt{case}(n, e_2, x \to e_3)}\ \textit{if}\ e_1 \overset{\rho}{=} n$$

For now, we leave `case#` stuck at unevaluated expressions, but come back to this issue later. Bolingbroke and Peyton Jones [1] show how this availability optimisation is easily implemented using a straightforward common-subexpression elimination in a strict core language.

Traditionally, this optimisation does not work across procedure boundaries, because the data flow within a function definition is hidden. Our new strictness domain exposes the relative evaluation order of function arguments across procedure boundaries; this information enables the *interprocedural* form of the optimisation. Consider a function $f(x, y)$ with effect $1 + x \cdot y$; this has two returning control-flow paths, one evaluating neither variable and one evaluating y after x. While f is not strict in x or y (nor jointly strict in x and y as in arms of a conditional) we do know that, given a call $f(e_1, e_2)$, then whenever e_2 is evaluated the thunk for e_1 will already have been forced. This allows us to optimise a call $f(x, \texttt{case}(x, 0, z \to z))$, logically $f(x, x - 1)$, to $f(x, \texttt{case\#}(x, 0, z \to z))$

Hence we are interested in partial order information "is-always-evaluated-before". Each effect ϕ defines a partial order $\prec_\phi$ on the set of effect variables X as follows.

Definition 5 (Variable Evaluation Order). *We say that a variable x_1 must*[6] *be evaluated before variable x_2 with respect to effect ϕ in DNF, denoted $x_1 \prec_\phi x_2$, iff (with $x \neq x_1, x \neq x_2$)*

$$\begin{aligned} x_1 \prec_{x \cdot \phi} x_2 &= x_1 \prec_\phi x_2 & \qquad x_1 \prec_1 x_2 &= \textit{true} \\ x_1 \prec_{x_1 \cdot \phi} x_2 &= \textit{true} & x_1 \prec_0 x_2 &= \textit{true} \\ x_1 \prec_{x_2 \cdot \phi} x_2 &= \textit{false} & x_1 \prec_{\phi_1 + \phi_2} x_2 &= x_1 \prec_{\phi_1} x_2 \wedge x_1 \prec_{\phi_2} x_2 \end{aligned}$$

For effects ϕ that are not in DNF, the relation is defined as:

$$x_1 \prec_\phi x_2 = x_1 \prec_{\mathit{dnf}(\phi)} x_2$$

For instance, the effect $x_1 \cdot x_2 + x_1 \cdot x_3$ captures the following order information:

$\prec$	x_1	x_2	x_3
x_1	–	Y	Y
x_2	N	–	N
x_3	N	N	–

as does $x_1 \cdot (x_2 + x_3)$. Note that in the case of 0 we can choose the variable evaluation order arbitrarily.

It is important to note that $\prec_\phi$ does not respect $\equiv$ (and hence is not a congruence for terms not in DNF), due to the behaviour of 0. For example,

[6] Only paths which can terminate are considered.

suppose we have code f with one path which evaluates first x and then y. This has effect $\phi = x \cdot y$ and so $x \prec_\phi y$ holds, but not $y \prec_\phi x$. Suppose now there is a definite loop before, or more problematically after, this code. Now its effect is $\phi' = 0 = \phi \cdot 0 = 0 \cdot \phi$ and note that both $x \prec_{\phi'} y$ and $y \prec_{\phi'} x$ hold. This appears paradoxical, in that code which evaluates x first and then y and then loops can be deduced to evaluate y before x! The resolution is that only paths which can return a result are considered by the $\prec_\phi$ relation; and using an incorrect order of evaluation on non-terminating paths does not matter (save for an implementation effect we explore in the next section). This effect also occurs if the code has multiple paths; the effect of 0 is to remove guaranteed non-terminating paths from consideration in the overall effect.

4.5 Transformation Soundness

There is a subtlety concerning the path 0 which we noted above. 0 represents a path which can never return, the archetypal example being a function call `loop()` given a definition `loop() = loop()`. While 0 behaves as an identity for $+$ and a (left and right) zero for $\cdot$ these algebraic properties which are fine for analysis need care when being used for optimisation.

This is related to partial versus total correctness: given function $f(x, y)$ having effect $x \prec_\phi y$ should not be simply read as "x is always evaluated before y", but more properly should be read as "x is always evaluated before y *whenever f returns*".

While the exact behaviour of code on non-terminating paths is not in general interesting, we must be careful that data-representation errors do not occur (these could replace non-termination with memory faults, or even seemingly valid answers). Consider again optimising a call $f(x, \texttt{case}(x, 0, z \to z))$ to $f(x, \texttt{case\#}(x, 0, z \to z))$ when we know $f(x, y)$ has effect ϕ and we have $x \prec_\phi y$. The problem is that f could have a definition such as $f(x, y) = \texttt{case}(y, f(x, y), z \to f(x, y))$ which would cause the potentially unevaluated thunk for x in the second argument of the call to be discriminated by `case#`. While this is clearly not a problem for unboxed values such as small integers and booleans (since the question is which of two infinite loops are taken), for values represented as pointers to code or to data this can spell memory errors or branches to arbitrary locations.

Formally, we model the issue with (i) an erroneous effect Err, that is generated when `case#` encounters an unevaluated expression, and (ii) a non-deterministic value n.

$$(\textsc{Err})\ \frac{}{\rho; \texttt{case\#}(e_1, e_2, x \to e_3) \overset{\textsc{Err}}{\rightarrowtail} \rho; \texttt{case}(n, e_2, x \to e_3)}\ \textit{if}\ e \overset{\rho}{\not\Downarrow} \qquad (\forall n)$$

A transformation that introduces `case#` necessarily (by soundness) avoids the (Err) transition on all terminating derivations (non-0 paths). Otherwise a change in semantics can be observed: the Err effect shows up in the trace of the transformed program, but not the original program. For non-terminating derivations, we distinguish between *fragile* and *robust* implementations.

Fragile implementations distinguish between the erroneous transition (e.g. yielding a crash) and non-termination. This is modelled by the supplementary law $\textsc{Err} \cdot 0 \not\equiv 0$ overriding the more general $\phi \cdot 0 \equiv 0$. Thus any (Err) transition, whether for a terminating or non-terminating derivation, violates the soundness of the optimisation. There are various ways to avoid (Err) on the 0 path for fragile implementations (e.g. dropping the "right zero" law), but we prefer to avoid the 0 path itself, analogous to traditional dataflow analysis, by adding a dummy return node to every loop that otherwise would not have one. The downside of avoiding the (Err) transition is of course the reduced opportunity for optimisations, which is why we turn to robust implementations.

Robust implementations are still modelled by $\textsc{Err} \cdot 0 \equiv 0$, and (Err) transitions in non-terminating derivations are not observable. This is for instance possible by ensuring that the payload of an unevaluated thunk is always interpretable as a value of its result type; hence the non-deterministic value n in rule (Err).

5 Related Work

Jensen [4] presents a strictness analysis based on strictness logic. His strictness language is perhaps the closest to ours, with polymorphic variables α and conditional strictness $\phi_2?\phi_1$ similar to respectively our parameters x and sequential composition $\phi_1 \cdot \phi_2$. However, as it lacks branching $(+)$ and 0, our novel optimisations do not apply.

We have only considered *flat* data, i.e. the natural numbers, where forcing the value also forces the component. Wadler [8] shows one way to extend strictness analysis to non-flat domains. A similar technique would apply to our domain.

Wansbrough [10] annotates function types with polymorphic "usage" annotations to identify thunks which are encountered at most once; these can be optimised to remove the "is-evaluated" test. It is appealing to speculate whether an extended effect system can capture this property too.

6 Conclusion and Future Work

We have expressed strictness as effects in a type-and-effect system which both adds insight into strictness properties and provides additional strictness optimisation opportunities.

There is a natural extension of our type-and-effect system to higher-order and polymorphic types (we need effect variables and an effect binding construct so that $\lambda x.x$ has effect $\forall \alpha . \alpha \xrightarrow[x]{x} \alpha$). With a subtyping rule, similar to (Coerce) in Section 2.2, the system becomes a conservative extension of polymorphic types, however it remains unclear whether the type-and-effect system has principal types, necessary for a type inference algorithm.

Acknowledgements Tom Schrijvers gratefully acknowledges funding for visiting the University of Cambridge from the Fund for Scientific Research – Flanders. The authors thank the anonymous reviewers for their helpful comments.

References

1. M. C. Bolingbroke and S. L. Peyton Jones. Types are calling conventions. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 1–12, New York, NY, USA, 2009. ACM.
2. G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *on Programs as data objects*, pages 42–62, New York, NY, USA, 1985. Springer.
3. L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Comput. Surv.*, 8(3):305–330, 1976.
4. T. P. Jensen. Inference of polymorphic and conditional strictness properties. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–221, New York, NY, USA, 1998. ACM.
5. S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* 2003.
6. J. Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, New York, NY, USA, 1993. ACM.
7. A. Mycroft. *Abstract interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.
8. P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation.* Ellis Horwood, 1987.
9. P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *FPCA*, volume 274 of *LNCS*, pages 385–407. Springer, 1987.
10. K. Wansbrough. Simple polymorphic usage analysis. Technical Report UCAM-CL-TR-623, Cambridge University Computer Laboratory, March 2005.