

Strictness Meets Data Flow

Tom Schrijvers¹ and Alan Mycroft²

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, initially at first order, then at higher order 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; without polymorphic effects principality is lost at higher order. However, adding both polymorphic effects and polymorphic type instantiation to restore principality exposes novel issues.

1 Introduction

Fosdick and Osterweil [2] 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 [3]. 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 [1, 4] or projections [6], 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 First-order 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 $f(x_1, \dots, x_k) = e$. Expressions e are function parameters x , function application $f(e_1, \dots, e_k)$, integer (natural number) constant 0, constructor application $\text{succ}(e)$ and case elimination $\text{case}(e_1, e_2, x \rightarrow e_3)$ (where either e_2 is returned if e_1 evaluates to 0, or e_3 is returned if e_1 evaluates to $\text{succ}(x)$).

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 $\tau_1, \dots, \tau_k \xrightarrow{\phi} \tau$ where τ_i are the argument types, τ the return type, and ϕ its effect.

An effect ϕ is either *parameter* x_i denoting the effect of evaluating the i th function argument x_i (arithmetic variables bound by **case** are effectively eager),

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. Polymorphism is considered in Section 6.

Source Language	Types and Effects
programs $p ::= d_1 \cdots d_n$	effects $\phi, \psi ::= x_i$
definitions $d ::= f(x_1, \dots, x_k) = e$	$ 0$
expressions $e ::= x$	$ 1$
$ f(e_1, \dots, e_k)$	$ \phi_1 \cdot \phi_2$
$ 0$	$ \phi_1 + \phi_2$
$ \text{succ}(e)$	value types $\tau ::= \text{Nat}$
$ \text{case}(e_1, e_2, x \rightarrow e_3)$	function types $\hat{\sigma} ::= \tau_1, \dots, \tau_k \xrightarrow{\phi} \tau$
	variable typing $\eta ::= \tau \ \& \ \phi$
$\phi_1 + \phi_2 \equiv \phi_2 + \phi_1$ (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

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 variable.

Effect Equivalence 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 $\{x_1, \dots, x_k\}$, 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 $\text{dnf}(\phi)$.*

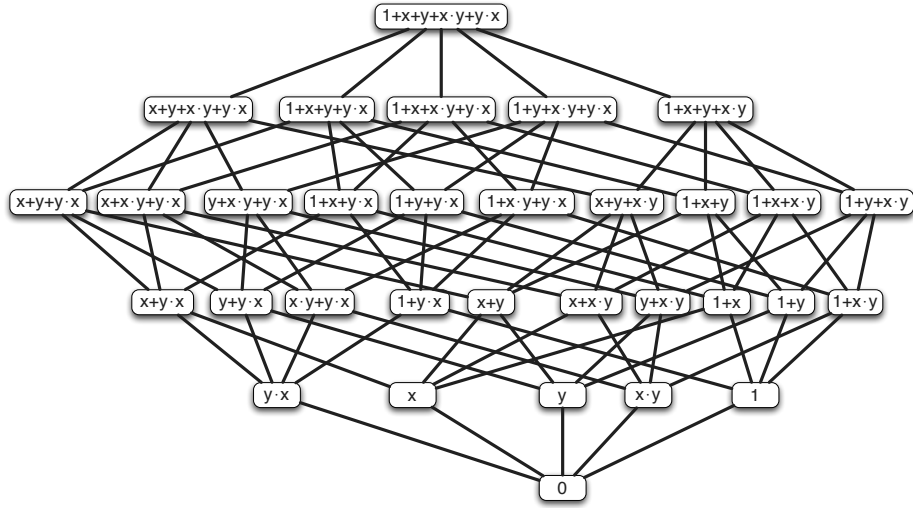


Fig. 2. The 32 different effects involving the two variables x and y

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 2). 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 $X = \{x_1, \dots, x_n\}$ as

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

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



Fig. 3. The effect domain ranging over a single effect variable x_1 .

blocks in the bitvector representation may be chosen arbitrarily. Figure 3 lists the distinct effects for $n = 1$ with their bitvector representation.

Correspondence to Regular Languages There is a surjective mapping $\llbracket \cdot \rrbracket$ from regular languages \mathcal{L} over the alphabet $A = \{a_1, \dots, a_n\}$ to effects ϕ ranging over parameters $X = \{x_1, \dots, x_n\}$.

$$\begin{array}{lll} \llbracket \emptyset \rrbracket = 0 & \llbracket \mathcal{L}_1 \cdot \mathcal{L}_2 \rrbracket = \llbracket \mathcal{L}_1 \rrbracket \cdot \llbracket \mathcal{L}_2 \rrbracket & \llbracket a_i \rrbracket = x_i \\ \llbracket \epsilon \rrbracket = 1 & \llbracket \mathcal{L}_1 \mid \mathcal{L}_2 \rrbracket = \llbracket \mathcal{L}_1 \rrbracket + \llbracket \mathcal{L}_2 \rrbracket & \llbracket \mathcal{L}^* \rrbracket = \mu\phi.(1 + \llbracket \mathcal{L} \rrbracket) \cdot \phi \end{array}$$

Note that, due to the addition of law (12), we do not need a Kleene star constructor as part of the effect language.

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 3. 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:

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

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

$$\begin{array}{c}
\boxed{\Gamma \vdash e : \tau \& \phi} \quad (\text{VAR}) \frac{}{\Gamma \vdash x : \tau \& \phi} \text{ if } (x : \tau \& \phi) \in \Gamma \\
\\
(\text{ZERO}) \frac{}{\Gamma \vdash 0 : \text{Nat} \& 1} \quad (\text{SUCC}) \frac{\Gamma \vdash e : \text{Nat} \& \phi}{\Gamma \vdash \text{succ}(e) : \text{Nat} \& \phi} \\
\\
(\text{FUNAPP}) \frac{\Gamma \vdash e_i : \tau_i \& \phi_i \quad (i \in 1..k)}{\Gamma \vdash f(e_1, \dots, e_k) : \tau \& \phi[\bar{\phi}_i/\bar{x}_i]} \text{ if } (f : \tau_1, \dots, \tau_k \xrightarrow{\phi} \tau) \in \Gamma \\
\\
(\text{CASE}) \frac{\Gamma \vdash e_1 : \text{Nat} \& \phi_1 \quad \Gamma \vdash e_2 : \tau \& \phi_2 \quad \Gamma, x : \text{Nat} \& 1 \vdash e_3 : \tau \& \phi_3}{\Gamma \vdash \text{case}(e_1, e_2, x \rightarrow e_3) : \tau \& \phi_1 \cdot (\phi_2 + \phi_3)} \\
\\
\boxed{\Gamma \vdash f(\bar{x}) = e} \quad (\text{DEF}) \frac{\Gamma, \bar{x} : \bar{\tau} \& \bar{x} \vdash e : \tau \& \phi}{\Gamma \vdash f(\bar{x}) = e} \text{ if } (f : \bar{\tau} \xrightarrow{\phi} \tau) \in \Gamma \\
\\
\boxed{\Gamma \vdash \bar{d}} \quad (\text{PROG}) \frac{\Gamma \vdash d_1 \quad \dots \quad \Gamma \vdash d_n}{\Gamma \vdash d_1 \dots d_n}
\end{array}$$

Fig. 4. Type-and-Effect Inference Rules

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 $f : \tau_1, \dots, \tau_k \xrightarrow{\phi} \tau$.

Figure 4 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 (FUNAPP) makes sure that the types of the arguments match the function typing in the environment.

Rules (ZERO) and (SUCC) cover the two 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).

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 $f : \mathbf{Nat} \xrightarrow{\phi} \mathbf{Nat}$ for any effect ϕ . Similarly, $f(x_1, x_2) = \mathbf{case}(x_1, x_2, y \rightarrow f(y, x_2))$ 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 classical 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:

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

but this is unnecessary here because (i) effects can express non-deterministic choice using $+$, and (ii) in the first-order setting, subeffecting only applies co-variantly 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 Hindley-Milner type inference

The type-and-effect system we have defined, and the higher-order extensions in Section 5, have 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:

$$\epsilon(\tau \ \& \ \phi) = \epsilon(\tau) \qquad \epsilon(\mathbf{Nat}) = \mathbf{Nat} \qquad \epsilon(\bar{\tau} \xrightarrow{\phi} \tau) = \bar{\tau} \rightarrow \tau$$

and lifted to environments Γ as usual.

Results A well-typing in the type-and-effect system is also a traditional well-typing. 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 (Soundness & Completeness).

$$(\forall e, \Gamma, \tau, \phi) \Gamma \vdash e : \tau \ \& \ \phi \Rightarrow \epsilon(\Gamma) \vdash_{HM} e : \epsilon(\tau)$$

$$(\forall \Gamma, e, \tau) \Gamma \vdash_{HM} e : \tau \Rightarrow (\exists \Gamma', \tau', \phi) \Gamma' \vdash e : \tau \ \& \ \phi \wedge \epsilon(\Gamma) = \Gamma' \wedge \epsilon(\tau) = \tau$$

Soundness follows from the fact that the erasure operator maps the rules of the type-and-effect system to those of traditional types.

3 Inference Algorithm

Figure 5 lists our first-order inference algorithm. The inference judgement for expressions is of the form $\Gamma \vdash^A e : \tau \ \& \ \phi \mid C$, 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 $\tau = \tau'$.⁴

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 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 $f(x_1) = x_1$ which gives effect equation $x_1 <: \phi$. As ϕ does not appear in the left-hand side of the constraint, the least fixpoint x_1 is immediately reached.

Example 2. Consider the function definition $g(x_1, x_2) = \mathbf{case}(x_1, x_2, y \rightarrow g(y, x_2))$ 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:

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

⁴ $\models C$ denotes that C is satisfiable, usually established by unification.

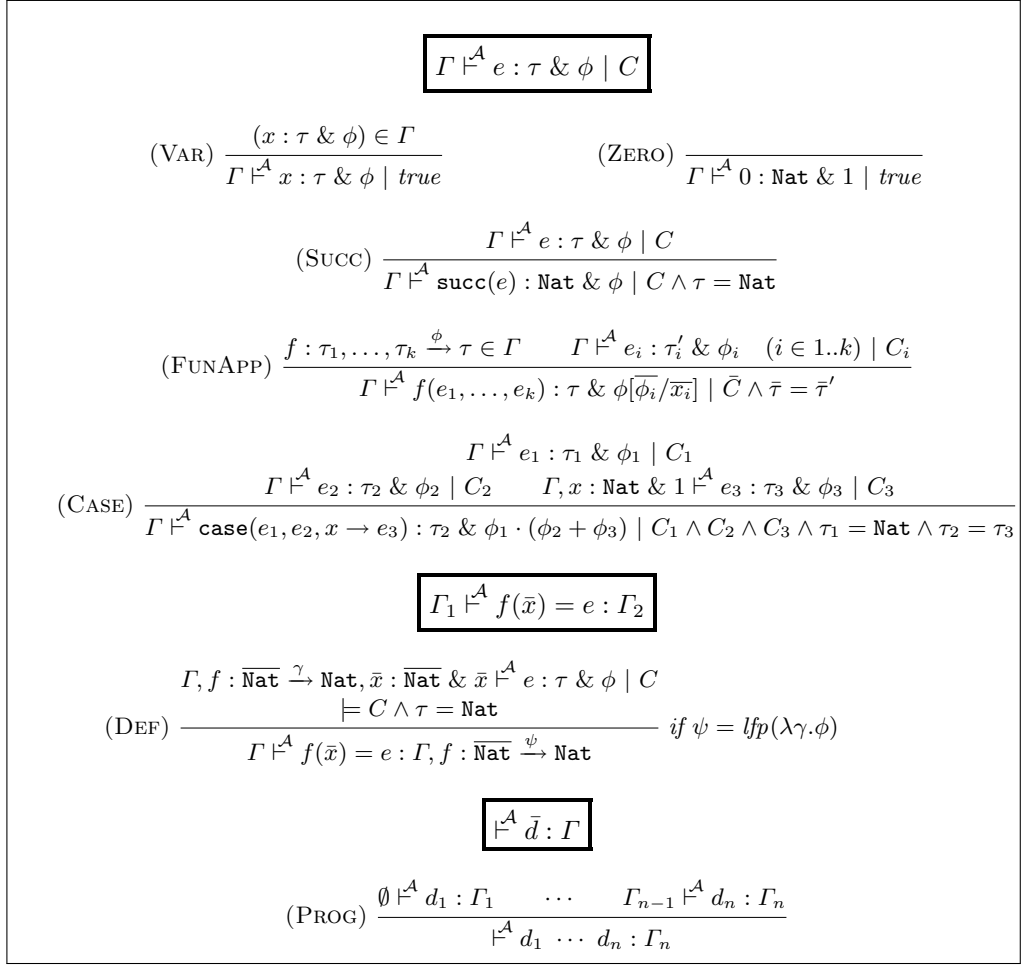


Fig. 5. Syntax-Directed Inference Algorithm

3.1 Properties

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

Theorem 4 (Principality). *If $\Gamma \vdash \bar{d}$ and $\vdash^A \bar{d} : \Gamma'$, 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 functions used in traditional strictness analysis. The abstraction relation α that maps our effects to Boolean functions.

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

Theorem 6 (Abstraction Soundness). *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.

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 3. The function $f(x_1, x_2) = \text{case}(x_1, 0, y \rightarrow x_2)$ has type

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

which provides no direct opportunity for strictness optimisation of x_2 . However, after inlining, strictness optimisation can be applied directly 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 [6].

A function of type $f : \tau_1, \dots, \tau_k \xrightarrow{\phi} \tau$ does not evaluate its i th argument (on any path which can return) if $x_i \notin \phi$. For instance, a function of type $f : \text{Nat} \xrightarrow{1} \text{Nat}$ does not evaluate its argument. Hence, the function definition can be rewritten from $f(\dots, x_{i-1}, x_i, x_{i+1}, \dots) = e$ to $f(\dots, x_{i-1}, x_{i+1}) = e$, and likewise the i th 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. $f(x) = \text{case}(x, f(x), y \rightarrow f(x))$.

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

An alternative and novel optimisation opportunity for non-strict functions consists of exploiting the relative ordering of evaluations. 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 $f(x, \text{case}(x, 0, z \rightarrow z))$, logically $f(x, x-1)$, to $f(x, \text{case}\#(x, 0, z \rightarrow z))$ where `case#` does not force its argument (i.e. reads the payload of its discriminant directly).

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

Definition 4 (Variable Evaluation Order). We say that a variable x_1 must be evaluated before variable x_2 with respect to effect ϕ in DNF, denoted $x_1 \prec_\phi x_2$, iff

$$\begin{aligned}
x_1 \prec_{\phi_1 + \phi_2} x_2 &= x_1 \prec_{\phi_1} x_2 \wedge x_1 \prec_{\phi_2} x_2 \\
x_1 \prec_{x_1 \cdot \phi} x_2 &= \text{true} \\
x_1 \prec_{x_2 \cdot \phi} x_2 &= \text{false} \\
x_1 \prec_{x \cdot \phi} x_2 &= x_1 \prec_\phi x_2 \quad (x \neq x_1, x \neq x_2) \\
x_1 \prec_0 x_2 &= \text{true} \\
x_1 \prec_1 x_2 &= \text{true}
\end{aligned}$$

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

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

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

	<		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_ϕ does not respect \equiv (and hence is not a congruence for terms not in DNF), due to the behaviour of 0. For example, 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_ϕ 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.

Optimisation and the empty path 0 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 $f(x, \mathbf{case}(x, 0, z \rightarrow z))$ to $f(x, \mathbf{case}\#(x, 0, z \rightarrow z))$ 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 $f(x, y) = \mathbf{case}(y, f(x, y), z \rightarrow f(x, y))$ which would cause the potentially unevaluated thunk for x in the second argument of the call to be discriminated by $\mathbf{case}\#$. 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.

There are various resolutions of this problem (e.g. changing the laws for 0 , e.g. changing code-generation so that the payload from an unevaluated thunk is a type-representation-correct value), but we note that functions lacking a return node (because all paths are recursive) are not very useful, and so here proceed by analogy of traditional dataflow analysis. There each node in a control-flow graph (CFG) is assumed “to have a path to it from the entry node” (natural enough) but also “to have a path from it to the exit node”. This latter property ensures that all loops have an exit and so invalid optimisations are not made on a path which must loop forever. In CFGs this property can be ensured by identifying nodes which cannot reach the exit node and adding conditional branches from them to the exit node. The analogous treatment here is to add returning, but never-taken, conditionals at the head of any function lacking a return path. Since such branches have no run-time effect, such new paths can safely be given any dataflow property and optimisation is helped by ascribing effects representing “evaluate all variables”. Unfortunately, our effects include order of evaluation of variables and it is unclear as how to pick the ‘best’ order to avoid losing optimisations on reachable code which calls such a function.

5 Higher-Order Functions

Most modern functional programming languages are not first-order but higher-order. They allow functions to be passed as arguments to other functions and to be returned as results from function calls.

For our higher-order source language, we essentially embed the typed λ -calculus in our expression language:⁵

$$\begin{array}{ll} \text{programs} & p ::= d_1 \cdots d_n \\ \text{definitions} & d ::= \mathbf{letrec} \ x = e \\ \text{expressions} & e ::= x \mid 0 \mid \mathbf{succ} \mid \mathbf{case}(e_1, e_2, x \rightarrow e_3) \mid e_1 \ e_2 \mid \lambda x. e \\ \text{types} & \hat{\tau} ::= \mathbf{Nat} \mid \hat{\tau} \xrightarrow{\phi} \hat{\tau} \end{array}$$

⁵ It is convenient to constrain the e in a definition to be of the form $\lambda y. e'$.

Value and function types are unified and effects can appear recursively in both argument and result types. For a type-and-effect $\hat{\tau} \& \phi$, we adopt the term *immediate* effect for ϕ , and we refer to the effects embedded in $\hat{\tau}$ as *latent* effects.

We now need to explain the new effect annotations on function types $\hat{\tau} \xrightarrow{\phi} \hat{\tau}$ more carefully. At first order function typings were just given as e.g. $f : \tau_1, \tau_2 \xrightarrow{y \cdot x} \tau_0$. This was satisfactory as there was no static nesting of functions and rules (FUNAPP) and (DEF) could simply look up a function to determine its parameter list, so given $f(x, y) = e$ the above annotation really represented De-Brujin-style “evaluate second parameter and then first parameter”. In the higher order case things are more complicated. Consider the similar $e = \lambda x. \lambda y. \text{case}(y, x, z \rightarrow x)$. We still wish to write $y \cdot x$ as its latent effect, but the binding of x and y must now be explicit as variables bound to e are ignorant of the context in which the **case** expression occurs. Accordingly we give e the type $\text{Nat} \xrightarrow{1} \text{Nat} \xrightarrow{y \cdot x} \text{Nat}$: the annotations above the arrow are the effects of calling the appropriate functional value (1 or $x \cdot y$), and those below the arrow are the names of the parameter to it. Of course, such types may be α -renamed at will.

5.1 Type-and-Effect Subtyping

We extend the subeffect relation to type-and-effects η , which allows us to not only vary the immediate effect, but also the latent effects:

$$\begin{aligned} (\text{NAT}) \quad & \frac{}{\text{Nat} \& \phi \prec: \text{Nat} \& \phi'} \text{ if } \phi \prec: \phi' \\ (\text{ARROW}) \quad & \frac{\hat{\tau}'_1 \& \phi' \prec: \hat{\tau}_1 \& \phi \quad \hat{\tau}_2 \& \phi \cdot \psi \prec: \hat{\tau}_2 \& \phi' \cdot \psi'}{\hat{\tau}_1 \xrightarrow{\psi} \hat{\tau}_2 \& \phi \prec: \hat{\tau}'_1 \xrightarrow{\psi'} \hat{\tau}'_2 \& \phi'} \text{ if } \phi \prec: \phi' \end{aligned}$$

In Rule (ARROW), the arrow effect ψ and the latent effects in the return type $\hat{\tau}_2$ may vary *co*-variantly, while the effects in the argument type $\hat{\tau}_1$ may vary *contra*-variantly.

More subtle is the issue of preceding effects. Consider an expression $e : (\text{Nat} \xrightarrow{x_1} \text{Nat}) \& x_1$. This expression’s immediate effect is to evaluate a parameter x_1 , before returning a function that evaluates x_1 again as a latent effect. Obviously, the second evaluation of x_1 is redundant, following equation 12. Hence, the type-and-effect of e is equivalent to $(\text{Nat} \xrightarrow{1} \text{Nat}) \& x_1$, if we take order of evaluation into account.

These observations are captured in Rule (ARROW), which combines the immediate effect ϕ and latent effect ψ into the immediate effect $\phi \cdot \psi$ of the argument type $\hat{\tau}_2$. This enables the preceding effects to be taken into account for the latent effects in $\hat{\tau}_2$.

$$\begin{array}{c}
\boxed{\hat{\Gamma} \vdash e : \hat{\tau} \& \phi} \qquad (\text{VAR}) \frac{}{\hat{\Gamma} \vdash x : \hat{\tau} \& \phi} \text{ if } (x : \hat{\tau} \& \phi) \in \hat{\Gamma} \\
\\
(\text{ZERO}) \frac{}{\hat{\Gamma} \vdash 0 : \text{Nat} \& 1} \qquad (\text{SUCC}) \frac{}{\hat{\Gamma} \vdash \text{succ} : \text{Nat} \xrightarrow{x} \text{Nat} \& 1} \\
\\
(\text{ABS}) \frac{\hat{\Gamma}, x : \hat{\tau}_1 \& x \vdash e : \hat{\tau}_2 \& \phi}{\hat{\Gamma} \vdash \lambda x. e : \hat{\tau}_1 \xrightarrow{\phi} \hat{\tau}_2 \& 1} \\
\\
(\text{APP}) \frac{\hat{\Gamma} \vdash e_1 : \hat{\tau}_1 \xrightarrow{\phi_2} \hat{\tau}_2 \& \phi_1 \quad \hat{\Gamma} \vdash e_2 : \hat{\tau}_1 \& \phi_3}{\hat{\Gamma} \vdash e_1 e_2 : \hat{\tau}_2 \& \phi_1 \cdot \phi_2[\phi_3/x]} \\
\\
(\text{CASE}) \frac{\hat{\Gamma} \vdash e_1 : \text{Nat} \& \phi_1 \quad \hat{\Gamma} \vdash e_2 : \hat{\tau} \& \phi_2 \quad \hat{\Gamma}, x : \text{Nat} \& 1 \vdash e_3 : \hat{\tau} \& \phi_3}{\hat{\Gamma} \vdash \text{case}(e_1, e_2, x \rightarrow e_3) : \hat{\tau} \& \phi_1 \cdot (\phi_2 + \phi_3)} \\
\\
(\text{COERCE}) \frac{\hat{\Gamma} \vdash e : \hat{\tau}_1 \& \phi_1}{\hat{\Gamma} \vdash e : \hat{\tau}_2 \& \phi_2} \text{ if } (\hat{\tau}_1 \& \phi_1) <: (\hat{\tau}_2 \& \phi_2) \\
\\
\boxed{\hat{\Gamma} \vdash \text{letrec } x = e} \qquad (\text{DEF}) \frac{\hat{\Gamma} \vdash e : \hat{\tau} \& \phi}{\hat{\Gamma} \vdash \text{letrec } x = e} \text{ if } (x : \hat{\tau} \& \phi) \in \hat{\Gamma} \\
\\
\boxed{\hat{\Gamma} \vdash \bar{d}} \qquad (\text{PROG}) \frac{\hat{\Gamma} \vdash d_1 \quad \dots \quad \hat{\Gamma} \vdash d_n}{\hat{\Gamma} \vdash d_1 \dots d_n}
\end{array}$$

Fig. 6. Monomorphic Higher-Order Type and Effect Rules

5.2 Higher-Order Type-and Effect-System

The higher-order type-and-effect system listed in Figure 6 is quite similar to the first-order system (Figure 4). There is only one notable change: adding the (COERCE) rule, which allows subtyping of type-and-effect.

Analogously to Section 2.3 this higher-order type-and-effect system is also a *conservative extension* of the higher-order simply typed λ -calculus. Erasing effects from a type-and-effect well-typing yields a simple well-typing. Vice versa, annotating every arrow in a simple well-typing with a unique parameter binder x and the chaos effect X^* of all parameters in scope, yields a type-and-effect well-typing.

The problem with this monomorphic type-and-effect system is a lack of principal types. There are two orthogonal reasons for this lack of principality. The first reason is the lack of principal types in the simply typed λ -calculus: $\text{letrec } id = \lambda x. x$ does not have a principal simple type. This lack of principal types is inherited through the conservative extension.

The second reason for the lack of principal types is due to the effect language. Consider the function $\text{letrec } iappi = \lambda f. \lambda x. succ(f(succ\ x))$. It has principal simple type $(\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$, yet there are four different minimal yet incomparable types:

$$\begin{array}{ll} (\text{Nat} \xrightarrow{0} \text{Nat}) \xrightarrow{1} (\text{Nat} \xrightarrow{0} \text{Nat}) & (\text{Nat} \xrightarrow{y} \text{Nat}) \xrightarrow{1} (\text{Nat} \xrightarrow{f \cdot x} \text{Nat}) \\ (\text{Nat} \xrightarrow{1} \text{Nat}) \xrightarrow{1} (\text{Nat} \xrightarrow{f} \text{Nat}) & (\text{Nat} \xrightarrow{y+1} \text{Nat}) \xrightarrow{1} (\text{Nat} \xrightarrow{f \cdot x + f} \text{Nat}) \end{array}$$

So principality is not inherited through the conservative extension, but destroyed by effects appearing in functional arguments (rank-2 types).

The next section attempts to restore principality by turning to polymorphism on both accounts: (i) in the type language and (ii) in the effect language.

6 Principality and Polymorphism

As we have seen in the previous section, the well-understood higher-order mono-variant type-and-effect system fails to produce principal types. In this section, we eradicate the non-principality by introducing two orthogonal forms of polymorphism well-known from the literature: *type polymorphism* to restore principality for the simply typed λ -calculus, and *effect polymorphism* to restore principality to effect systems.

The solutions seem simple enough. We extend the language of types and effects with polymorphic type variables β and polymorphic effect variables ξ , and definitions get type schemes that quantify over these variables.

$$\begin{array}{l} \text{effects } \phi ::= \xi \mid x \mid 0 \mid 1 \mid \phi_1 + \phi_2 \mid \phi_1 \cdot \phi_2 \\ \text{types } \hat{\tau} ::= \beta \mid \text{Nat} \mid \hat{\tau}_1 \xrightarrow{\phi} \hat{\tau}_2 \\ \text{schemas } \hat{\sigma} ::= \forall \xi. \hat{\sigma} \mid \forall \beta. \hat{\sigma} \mid \hat{\tau} \end{array}$$

Only two rules of the type-and-effect system are affected, in the obvious manner:

$$\text{(VAR)} \quad \hat{\Gamma} \vdash x : \hat{\tau}[\bar{\phi}/\bar{\xi}][\bar{\tau}/\bar{\beta}] \ \& \ \phi \ \text{if } (x : \forall \bar{\beta}. \forall \bar{\xi}. \hat{\tau} \ \& \ \phi) \in \hat{\Gamma}$$

$$\text{(DEF)} \quad \frac{\hat{\Gamma} \vdash e : \hat{\tau} \ \& \ \phi}{\hat{\Gamma} \vdash \text{letrec } x = e} \ \text{if } (x : \forall \bar{\beta}. \forall \bar{\xi}. \hat{\tau} \ \& \ \phi) \in \hat{\Gamma}$$

Surprisingly, we find polymorphism to be far less trivial than expected. Firstly, interpreted naively, both substitutions in (VAR) can give rise to (invalid) *scope extrusion* of effect parameters. Secondly, fixpoint equations for recursive functions require special attention. This section presents fixes for both these issues, but does not establish principality.

We now discuss the two forms of polymorphism independently, retaining only the forms with β or ξ .

6.1 Type Polymorphism

Polymorphic type variables, ranged over by β , resolve the non-principality due to simple types. So the identity function has principal type $\forall\beta.\beta \xrightarrow{x} \beta$. However, the naive adoption of ‘type polymorphism by substitution’ in our type-and-effect system leads to two serious issues:

Scope Extrusion Surprisingly, the instantiation of type variables may lead to scope extrusion of effect parameters. Consider the function $app0 = \lambda f.f\ 0$. It would seem that $\forall\beta.(\text{Nat} \xrightarrow{x} \beta) \xrightarrow{f} \beta$ is a valid type for $app0$. Yet consider what type for $(app0\ p)$ follows from this assumption, where p has type $\text{Nat} \xrightarrow{x} \text{Nat} \xrightarrow{x \cdot y} \text{Nat}$. A direct substitution of β yields $\text{Nat} \xrightarrow{x \cdot y} \text{Nat}$, but x is not in scope here and the type is not closed! The scope extrusion renders the instantiation invalid. In contrast, if we choose the monomorphic type $(\text{Nat} \xrightarrow{x} \text{Nat} \xrightarrow{x \cdot y} \text{Nat}) \xrightarrow{f} (\text{Nat} \xrightarrow{y} \text{Nat})$ for $app0$, then the application $app0\ p$ does have the valid type $\text{Nat} \xrightarrow{y} \text{Nat}$.

Unresolved Non-Principality Polymorphic type variables do not always resolve the non-principality issue. Consider the function $letrec\ choose = \lambda x.\lambda y.case(0, x, z \rightarrow y)$ with the seemingly principal polymorphic type-and-effect $\forall\beta.\beta \xrightarrow{x} \beta \xrightarrow{x+y} \beta$. The erased form of this type-and-effect is indeed the principal traditional type. However, it fails to capture some monomorphic type-and-effects like $(\text{Nat} \xrightarrow{z} \text{Nat}) \xrightarrow{x} (\text{Nat} \xrightarrow{z} \text{Nat}) \xrightarrow{x+y} (\text{Nat} \xrightarrow{x+y \cdot z} \text{Nat})$. For this to be an actual instance of the polymorphic type-and-effect, β must be simultaneously instantiated to three different types $(\text{Nat} \xrightarrow{z} \text{Nat})$, $(\text{Nat} \xrightarrow{z} \text{Nat})$ and $(\text{Nat} \xrightarrow{x+y \cdot z} \text{Nat})$, which is of course not possible because these types are not equivalent.

The problem in both cases is that simple substitution for a polymorphic type variable β requires that its different occurrences to be instantiated with the same type structure and the same effect structure. The former is essential for traditional well-typing, but the latter is an artificial restriction merely imposed by the form of our polymorphic type-and-effect language. This artificial restriction means that our polymorphic types are not adequate for resolving non-principality.

Polymorphic Effect Structures In order to adequately adapt polymorphic type variables β to our setting, we annotate them with an effect structure δ . Thus the notation $\beta\{\delta\}$ provides us with a separate name β for the type structure and δ for the effect structure. Hence, two type-and-effects $\beta\{\delta_1\}$ and $\beta\{\delta_2\}$ have the same traditional type, but (potentially) different latent effects. Our language of

types and effects becomes:

$$\begin{array}{ll}
\text{effect structures } \delta & ::= \zeta[\bar{\phi}/\bar{x}] \mid \delta_1 + \delta_2 \mid \phi \cdot \delta \\
\text{types } \hat{\tau} & ::= \beta\{\delta\} \mid \dots \\
\text{schemas } \hat{\sigma} & ::= \forall\beta.\hat{\sigma} \mid \forall\zeta.\hat{\sigma} \mid \dots
\end{array}$$

Here ζ denotes a polymorphic effect structure variable,⁶ $\delta_1 + \delta_2$ represents a non-deterministic choice of effect structures, and $\phi \cdot \delta$ represents an effect structure δ preceded by an effect ϕ (which is immediate wrt. δ). Note that effect structures δ are quite different from effects ϕ ; the effect structure is a place holder for an as of yet unspecified concrete structure⁷ $\Delta ::= \star \mid \Delta_1 \xrightarrow{\phi} \Delta_2$ with zero or more latent effects and parameter bindings.

A concrete effect structure Δ and a traditional type τ form a type-and-effect $\hat{\tau} = \tau\{\Delta\}$, if they are in structural correspondence:

$$\text{Nat}\{\star\} \equiv \text{Nat} \quad (\tau_1 \rightarrow \tau_2)\{\Delta_1 \xrightarrow{\phi} \Delta_2\} \equiv \tau_1\{\Delta_1\} \xrightarrow{\phi} \tau_2\{\Delta_2\}$$

Hence, the instantiation of a type structure β to a traditional type τ and its effect structure δ to a concrete structure Δ must happen simultaneously: if either structure is known, the other is fixed as well.

Consider again the function *choose*. Its principal type is properly expressed as $\forall\beta.\forall\zeta_1.\zeta_2.\beta\{\zeta_1\} \xrightarrow{1/x} \beta\{\zeta_2\} \xrightarrow{x+y/y} \beta\{x \cdot \zeta_1 + y \cdot \zeta_2\}$. We obtain the earlier monomorphic type of *choose* from the polymorphic one by the simultaneous substitution $[(\text{Nat} \rightarrow \text{Nat})/\beta, (\star \xrightarrow{1/x} \star)/\zeta_1, (\star \xrightarrow{z/y} \star)/\zeta_2]$.

The role of the parameter substitution construct $[\bar{\phi}/\bar{x}]$ on effect structure variable ζ is to avoid scope extrusion. For instance, a proper polymorphic type of *app0* is $\forall\beta.\forall\zeta.(\text{Nat} \xrightarrow{1/x} \beta\{\zeta\}) \xrightarrow{f} \beta\{\zeta[1/x]\}$. The type of *p* above matches that of the argument in the polymorphic type through the substitution $[(\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat})/\beta, (\star \xrightarrow{1/x} \star \xrightarrow{x \cdot y/y} \star)/\zeta]$. Thanks to the parameter substitution the return type $(\text{Nat} \xrightarrow{x \cdot y/y} \text{Nat})[1/x] \equiv (\text{Nat} \xrightarrow{y/y} \text{Nat})$ is closed.

The extension of the equivalence and subtyping relations of the new type structure and effect structure features is a straightforward promotion of theorems in the monomorphic system to axioms in the polymorphic setting. For lack of space we omit them.

6.2 Effect Polymorphism

Polymorphic effect variables, ranged over by ξ , restore principality of effects. As with type polymorphisms, the instantiation of effect variables may cause scope

⁶ Possibly annotated with a formal parameter substitution explained shortly

⁷ Not part of the source language.

extrusion of parameters. Parameter substitution (here $\xi[\bar{\phi}/\bar{x}]$) again resolves that problem. For instance, *iappi* has principal type:

$$\forall \xi. (\text{Nat} \xrightarrow{y} \text{Nat}) \xrightarrow{1} (\text{Nat} \xrightarrow{f \cdot \xi[x/y]} \text{Nat})$$

The parameter substitution $[x/y]$ on the second occurrence of the polymorphic effect variable is vital to avoid scope extrusion of effect variables. Consider the type of $(iappi\ g)$ where $g : \text{Nat} \xrightarrow{y} \text{Nat}$ when we omit the parameter substitution.

The polymorphic effect variable ξ is instantiated to y , and we obtain $(iappi\ g) : (\text{Nat} \xrightarrow{y} \text{Nat}) \& 1$. Here the parameter y has escaped its scope and the type has wrongly become open. In the presence of the substitution, the resulting latent effect is $y[x/y] \equiv x$, and y does not escape its scope.

Again the necessary equivalence and subtyping axioms follow from theorems in the monomorphic system, but are omitted for lack of space.

Lack of Fixpoints It turns out that the polymorphic effect language no longer guarantees the existence of fixpoints. Take the recursive function $\lambda f. case(0, 0, z \rightarrow f(g\ f))$ with type $(\text{Nat} \xrightarrow{x} \text{Nat}) \xrightarrow{\phi} \text{Nat}$ where ϕ is a solution of the equation $\phi \equiv 1 + f \cdot \xi[\phi/x]$. However, computing the least fixpoint iteratively, starting from 0, diverges:

$$\begin{aligned} \phi_0 &\equiv 0 \\ \phi_1 &\equiv 1 + f \cdot \xi[0/x] \\ \phi_2 &\equiv 1 + f \cdot \xi[1/x] + f \cdot \xi[\xi[0/x]/x] \\ \phi_3 &\equiv 1 + f \cdot \xi[\xi[1/x]/x] + f \cdot \xi[\xi[\xi[0/x]/x]/x] \\ &\dots \end{aligned}$$

whereas we do know that any monomorphic instantiation will converge. In order to resolve the issue, we propose using an explicit object-level fixpoint construct $\mu\ \gamma.\phi$ for the effect language, with main equivalence axiom:

$$(\text{UNFOLD})\ \mu\ \gamma.\phi \equiv \phi[(\mu\ \gamma.\phi)/\gamma]$$

6.3 Open Problems

In the above, we have studied two forms of polymorphism to recover from non-principality. In the combined system, we define a new ordering relation \prec : that combines polymorphic instantiation and subtyping:

$$\frac{\kappa(\hat{\tau}_1) \prec: \hat{\tau}_2}{\hat{\tau}_1 \prec: \hat{\tau}_2} \text{ for some } \kappa$$

where the substitution κ instantiates polymorphic type variables, effect structures and polymorphic effect variables.

We assume that all higher-order programs have a principal typing in the polymorphic type-and-effect system according to the \prec : ordering, and have found no counter-examples. However proving principality remains an open problem.

A second major open issue is the decidability of finding the principal type, if it exists. An important issue here is whether polymorphic effect recursion affects decidability, just as polymorphic type recursion does. If the principal type does not exist, the problem of finding it is not decidable, or the cost is simply prohibitive, then an effective approximative algorithm should be developed.

7 Conclusion

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. Our monomorphic first- and higher-order type systems are well-understood, although the latter suffers from a lack of principality. Addressing the principality of the higher-order system with polymorphism leads to novel issues. As for many other type-and-effect systems, it remains an open problem whether our polymorphic higher-order type-and-effect system exhibits principal types and whether they can be computed by an inference algorithm.

We have only considered *flat* data, i.e. the natural numbers, where forcing the value also forces the component. Wadler [5] shows one way to extend strictness analysis to non-flat domains. A similar technique would apply to our domain.

Wansbrough [7] 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.

Acknowledgements Tom Schrijvers gratefully acknowledges funding for visiting the University of Cambridge from the Fund for Scientific Research – Flanders.

References

1. 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.
2. Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Comput. Surv.*, 8(3):305–330, 1976.
3. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. 2003.
4. Alan Mycroft. *Abstract interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.
5. Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation*. Ellis Horwood, 1987.
6. Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In Gilles Kahn, editor, *FPCA*, volume 274 of *LNCS*, pages 385–407. Springer, 1987.
7. Keith Wansbrough. Simple polymorphic usage analysis. Technical Report UCAM-CL-TR-623, Cambridge University Computer Laboratory, March 2005.