Data-flow analyses as effects and graded monads

Andrej Ivašković
Department of Computer Science and Technology, University of Cambridge, UK
andrej.ivaskovic@cst.cam.ac.uk

Alan Mycroft
Department of Computer Science and Technology, University of Cambridge, UK
alan.mycroft@cst.cam.ac.uk

Dominic Orchard
School of Computing, University of Kent, UK
d.a.orchard@kent.ac.uk

Abstract

In static analysis, two frameworks have been studied extensively: monotone data-flow analysis and type-and-effect systems. Whilst both are seen as general analysis frameworks, their relationship has remained unclear. Here we show that monotone data-flow analyses can be encoded as effect systems in a uniform way, via algebras of transfer functions. This helps to answer questions about the most appropriate structure for general effect algebras, especially with regards capturing control-flow precisely. Via the perspective of capturing data-flow analyses, we show the recent suggestion of using *effect quantales* is not general enough as it excludes non-distributive analyses e.g., *constant propagation*. By rephrasing the McCarthy transformation, we then model monotone data-flow effects via *graded monads*. This provides a model of data-flow analyses that can be used to reason about analysis correctness at the semantic level, and to embed data-flow analyses into type systems.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases data-flow analysis, effect systems, graded monads, correctness

Supplement Material Code and additional proofs: `https://doi.org/10.5281/zenodo.3784967`

Funding *Andrej Ivašković*: funded by Trinity College, Cambridge (Internal Graduate Scholarship)
Dominic Orchard: funded in part by EPSRC grant EP/T013516/1

1 Introduction

Static program analysis is the bedrock of optimising compilation, extracting program properties from syntax to inform semantics-preserving program transformations. Throughout the history of program analysis it has been repeatedly noticed that various analyses have similar forms and can thus be unified into more general frameworks. Notably, the early *data-flow analyses* performed on control-flow graphs (e.g., for live variables, available expressions, reaching definitions etc.) were unified by the notion of *monotone data-flow frameworks* [8] (Khedker et al. [10] give a wider perspective). Such analyses are formalised as scanning program statements forwards or backwards to obtain data-flow equations over some algebraic structure, which are then solved. Another major class of analyses are *effect systems* [5, 7, 15, 24], typically applied in a functional setting (but also notably for Java's checked exceptions). Effect systems typically augment type systems with information about possible side-effects, drawn from a particular algebraic structure. Such approaches evolved into a framework for general static analysis [16]. Another general class of static analysis is *abstract interpretation* given by Galois connections or related structures [3], though this is not our focus here.

Despite claims of effect systems' generality, it has been unclear whether they have sufficient expressive power to capture classical data-flow analyses, due in part to the functional-style bias of effect systems but also due to a lack of clarity about how effect systems interact with control-

flow. Various approaches have developed effect-system-like systems for capturing particular data-flow analyses, but typically in an *ad hoc* manner. For example, Nielson, Nielson, and Hankin [17] presented an effect-system-like annotated type system for reaching definitions analysis, but the approach was not clearly linked to a general algebraic characterisation of effects seen elsewhere. Laud et al. [11] introduced several type systems that represent data-flow analyses, but these are not effect systems and the approach is not unified.

In this work, we study the general relationship of dataflow frameworks to effect systems, and through this investigate the most appropriate algebraic characterisation of effects to capture known analyses in a uniform way. While Gifford-Lucassen-style [5] effect annotations were originally seen as mere subsets of the space of possible effect operations along with a single composition operator, Amtoft and the Nielsons [1] showed how distinct sequencing and alternation operators for composing effects gave better expressivity, capturing various other analyses. Recently, Katsumata [9] and Orchard et al. [20] linked effect systems to the mathematical notion of *graded monads*, using graded monads to model languages with effect systems. The graded monad model characterises the algebraic structure of effect systems by the structure of its *grades* which constrain the model of a computation's side effects. In this setting, Katsumata offers the most-general framework for effect systems: an *effect algebra* is a pre-ordered monoid $(D, \sqsubseteq, \rhd, 1)$, where $(D, \sqsubseteq)$ is a pre-ordered set and $(D, \rhd, 1)$ a monoid with $\rhd$ monotonic with respect to $\sqsubseteq$ [9]. Binary least upper bounds on D, if they exist, give a natural (if partial) alternation operator. Gordon [6] by contrast aims, in recent work, at a more precise axiomatisation using *effect quantales* which enforce composition and alternation to be total, if necessary by adding an additional top (or error) element to D, and adding distributivity requirements. Mycroft et al. [15] also split effect algebras into separate operators for sequencing and alternation, with a graded monad model.

Two questions arise. Firstly, how related are the theories of data-flow analysis and effect systems, and their interpretation as graded monads? Secondly, what is the most natural structure for an effect algebra that covers common analyses?

Contributions and structure Section 2 begins by summarising various background material about data-flow analyses and effect systems. We then contribute three main results:

1. We show that monotone data-flow frameworks can be captured via a kind of effect system on control-flow graphs (CFGs) with effect algebras of transfer functions (Section 3). The approach ends up resembling Kam and Ullman's monotone data-flow analysis frameworks. The novelty is that the approach unifies several classical data-flow analyses.
2. We adapt McCarthy's transformation [12] to translate the CFG-effect system of Section 3 into a *graded monad* rendering of effect systems (Section 4). This gives a semantic model equipped with data-flow analysis information which can be used to reason about analysis correctness or to capture dataflow as types, which we demonstrate via a Haskell encoding.
3. We discuss how effect quantales are too restrictive to capture non-distributive data-flow analyses such as constant propagation (Section 5).

There are several interesting lines of further work that follow from the perspective of this paper. For example, computational complexity of data-flow analysis algorithms is well understood and results from this field may provide valuable insight in constructing efficient type-and-effect inference algorithms. We also aim to contribute towards finding a "best" model for effect algebras – one that imposes just enough restrictions that every static analysis can be modelled, while disallowing models which correspond to no (known) static analysis.

2 Background

2.1 Analysis structure: partial orders and lattices

Program analysis generally captures program properties as elements of a partially ordered set (*poset*). Often this poset forms a complete lattice, but this places strong requirements on the existence of least upper and greatest lower bounds, which are not always needed or desired. For example, in type inference we may infer that two expressions e_1 and e_2 have respective types Int and Bool, but then say that a conditional selecting between e_1 and e_2 is ill-typed. There seems to be a tacit understanding that static type inference is usually partial while static determination of other properties is total – perhaps because we are happy for a program to be rejected as ill-typed, but not for a compiler to reject our program just because a static analysis says it is unfit for a given optimisation. In general this distinction between partial and total analyses affects 'formal presentation' more than 'conceptual understanding' as we can make any partial analysis total by adding a $\top$ element to its poset of values.

A poset $(D, \sqsubseteq)$ is a set D with a reflexive, antisymmetric and transitive relation $\sqsubseteq$. Given two posets, $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ then their product $D_1 \times D_2$ has the induced product order: $(x, y) \sqsubseteq (x', y')$ whenever $x \sqsubseteq_1 x'$ and $y \sqsubseteq_2 y'$. Similarly, given any set X then $X \to D$ becomes a poset with induced ordering $f \sqsubseteq g$ whenever $\forall x \in X. f(x) \sqsubseteq g(x)$.

A poset $(D, \sqsubseteq)$ is a *(bounded) join-semilattice* if all finite (including empty) subsets $X \subseteq D$ have a least upper bound with respect to $\sqsubseteq$. It is a *(bounded) lattice* if such subsets also have a greatest lower bound. It is a *complete lattice* if all subsets have least upper bounds and greatest lower bounds. We write $\bot$ for $\bigsqcup\{\}$ and $\top$ for $\bigsqcup D$ when these exist. Much work on program analysis is done on posets of *finite height* (every totally ordered subset is finite) so completeness adds no additional requirements.

A join-semilattice is often axiomatised via an operator $(D, \sqcup)$ because this gives an algebraic characterisation; the relation $\sqsubseteq$ can be recovered by taking $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.

For data-flow analysis of Turing-complete languages we generally need $(D, \sqsubseteq)$, or $(D, \sqcup)$ to be *bounded* or *pointed*, i.e. to have a least element $\bot$ which can represent the data-flow values resulting from a non-terminating expression, and also serves as the initial value for a Tarski fixed-point iteration when solving data-flow equations.

2.2 Control-flow graphs

Classical compiler optimisations usually deal with simple imperative programs, represented as *control-flow graphs* (CFGs). Here statements S appearing within the flow graph and possibly containing branches to labels ℓ are given by:

$$
\begin{array}{lcll}
v & ::= & X \mid k & \text{(syntactic values)} \\
e & ::= & v \mid v_1 \ op \ v_2 & \text{(expressions)} \\
S & ::= & X := e;\ \mathsf{goto}\ \ell \mid \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else\ goto}\ \ell'' \mid \mathsf{halt}\ v & \text{(statements)}
\end{array}
$$

where k are assumed to be integers, *op* ranges over arithmetic operators ($+$, $-$, $\times$ etc.), and X ranges over *Vars*, a finite set of integer-valued mutable variables.

A CFG $(N, E \subseteq N \times N, \mathcal{L} : N \to S)$ is a directed graph whose nodes N are labelled with 3-address arithmetic and control-flow statements. We use n (and ℓ when thinking of a node as a label) to range over N. As usual, we write $succ(n)$ and $pred(n)$ for the sets of E-successors and E-predecessors of n, and require the number of successors of a node to respect the labelling $\mathcal{L}$. We write $(\ell : S)$ to indicate that node ℓ is labelled with a given statement or, in programming terms, that statement S has label ℓ.

2.3 Classical data-flow analysis

Data-flow analysis refers to static analysis approaches commonly used in optimising compilers. These analyses infer facts about how data is used in the program, including constant propagation, live variables and pointer analysis.

Liveness In a CFG, a variable x is *live* at node n if there is a (possibly infeasible) path of edges starting at n along which the value of X is read before being written to. The sets of variables live on entry and exit of n respectively satisfy the following data-flow equations:

$$\mathit{LiveIn}(n) = (\mathit{LiveOut}(n) \setminus \mathit{LiveKill}(n)) \cup \mathit{LiveGen}(n) \qquad \mathit{LiveOut}(n) = \bigcup_{s \in \mathit{succ}(n)} \mathit{LiveIn}(s)$$

Sets $\mathit{LiveKill}(n)$ and $\mathit{LiveGen}(n)$ are determined by the statement at node n. For statement $X := e$ they are $\mathit{LiveKill}(n) = \{X\}$ and $\mathit{LiveGen}(n) = \mathit{fv}(e)$ (the set of free variables in e). For halt v and if $v \geq 0$ they are $\mathit{LiveKill}(n) = \emptyset$ and $\mathit{LiveGen}(n) = \mathit{fv}(v)$.

We consider $\mathit{LiveIn}(n)$ to be the set of live variables just before the statement at node n, and $\mathit{LiveOut}(n)$ to be the live variables immediately after this statement. The notation $\mathit{Live}(n)$, $\mathit{gen}(n)$, and $\mathit{kill}(n)$ are used as synonyms for $\mathit{LiveIn}(n)$, $\mathit{LiveGen}(n)$, and $\mathit{LiveKill}(n)$.

Monotone data-flow analysis frameworks Liveness, along with several other analyses, can be seen as examples of Kam and Ullman's *monotone data-flow analysis frameworks* [8]. Roughly speaking, a monotone data-flow analysis framework[1] *instance* is specified by a lattice $(\mathit{DFValues}, \sqcup)$ with:

- the set $\mathit{DFValues}$ of all possible data-flow values with a *least element* $\bot$;
- the direction of the analysis, *forwards* or *backwards* (liveness is backwards since $\mathit{LiveOut}$ is calculated from successors);
- Gen and Kill sets for every statement;
- the *merge* operation $\sqcup$ (for liveness and reaching definitions this is $\cup$, whereas for available expressions and very busy expressions it is $\cap$).

As with liveness, such instances give a set of equations whose solutions give the data-flow values at every node in the CFG. An exception is made in the cases of incoming data-flow values for entry nodes in forwards analysis and outgoing data-flow in exit nodes in backwards analysis – they do not depend on other data-flow values, they are instead equal to the *boundary information* (BI, typically $\bot$ or $\top$). When there are multiple solutions, we take the *least* one (which exists because of the lattice assumption and the existence of $\bot$ in $\mathit{DFValues}$). An iterative *work-list algorithm* is used to compute data-flow values at every node.

Every node in a CFG determines a *transfer function* (or *flow function*) from the set of data-flow values at one end of the node to that at the other end (In to Out for forwards analyses and Out to In for backwards analyses). Transfer functions propagate data-flow values around a program. For backwards analyses, the transfer function ϕ satisfies $\mathit{DF}_{\mathit{In}}(n) = \phi(\mathit{DF}_{\mathit{Out}}(n))$; for forward analyses $\mathit{DF}_{\mathit{Out}}(n) = \phi(\mathit{DF}_{\mathit{In}}(n))$ (where $\mathit{DF}_{\mathit{In}}$ and $\mathit{DF}_{\mathit{Out}}$ map nodes to the data-flow values at entry and exit, like LiveIn and $\mathit{LiveOut}$ previously).

We can extend the idea of transfer functions for single statements to sequences of statements (or paths in a CFG). Consider, for example, statements S_1 and S_2 with associated

[1] The original work calculated maximal fixed points by iterating from a $\top$ value. We use the dual formulation (least fixed points and $\bot$ value). Our data-flow examples use complete lattices.

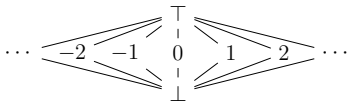

Figure 1 Lattice of integers $\mathbb{Z}_{\perp}^{\top}$ with $\top$ and $\perp$ whose product lattice is the lattice of data-flow values in constant propagation.

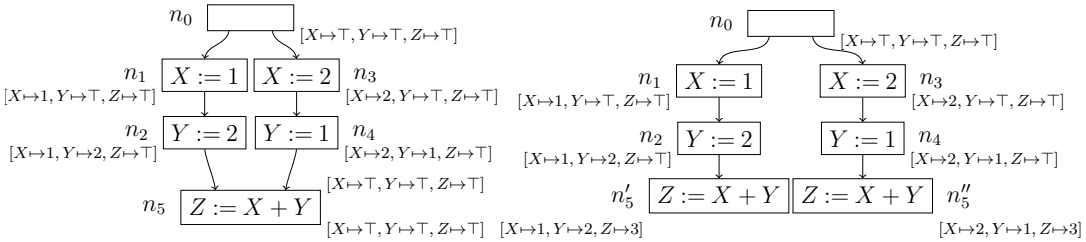

Figure 2 Non-distributivity of constant propagation. Assume data-flow into n_0 is such that multiple values can be associated with each of X, Y, Z and so these are mapped to $\top$. Non-distributivity manifests at node n_5 in the CFG on the left: whichever execution path is taken after n_0, Z has value 3 at n_5, but constant-propagation analysis gives $Z \mapsto \top$. Splitting the statement into two and performing constant-propagation analysis on these paths separately (as in the CFG on the right) gives $Z \mapsto 3$ which is more precise than $Z \mapsto \top$ in the left CFG, violating distributivity.

transfer functions ϕ_1 and ϕ_2. Then the transfer function for the sequence 'S_1 then S_2' is $\phi_1 \circ \phi_2$ for a backwards analysis, whereas for a forwards analysis it is $\phi_2 \circ \phi_1$ (this reverse composition is natural for forwards analysis, since $\phi_2 \circ \phi_1$ first applies ϕ_1 to the input, then applies ϕ_2 to the result).

Data-flow analyses may also have a notion of *distributivity* relating to the merging operator. A forwards or backwards analysis is *distributive* if it satisfies the following corresponding property for all nodes n:

$$DF_{Out}(n) = \bigsqcup_{n' \in pred(n)} \phi_n(DF_{Out}(n')) \quad (forward) \qquad DF_{In}(n) = \bigsqcup_{n' \in succ(n)} \phi_n(DF_{In}(n')) \quad (backward)$$

where ϕ_n is the transfer function for node n. Live variable analysis is a distributive analysis.

Constant propagation Some data-flow analyses are not distributive. One such example is *constant propagation*: a forwards analysis that associates with each program point a mapping (ranged over by s here) from variables to data-flow values which are either an integer or one of two special symbols $\perp$ or $\top$. The mapping $X \mapsto \top$ means that variable X potentially takes multiple values and so is not (known to be) a constant, whereas $Y \mapsto \perp$ means that the value of variable Y has not been explored yet in the analysis (this is needed for loops where the analysis uses fixed-point iteration). For integer variables, this gives a lattice of data-flow values of integers, along with $\perp$ and $\top$, shown in Figure 1.

This lattice naturally gives rise to a lattice of mappings which is just a *product lattice* with the subtlety that if one variable maps to $\perp$ then all do (a so-called $\perp$-*coalesced product*), with the partial order $\sqsubseteq$ lifted to the product space, e.g. $[X \mapsto 1,\ Y \mapsto 5] \sqsubseteq [X \mapsto 1,\ Y \mapsto \top]$ but $[X \mapsto 1,\ Y \mapsto 5] \not\sqsubseteq [X \mapsto 3,\ Y \mapsto 5]$. The formula $s_1 \sqsubseteq s_2$ can be read as 's_1 is more precise than s_2'. The result of the analysis should be a least solution of the data-flow equations. Merging is via least upper bounds ($\sqcup$) taken component-wise, e.g. $[X \mapsto 1,\ Y \mapsto 3] \sqcup [X \mapsto 2,\ Y \mapsto 3] = [X \mapsto \top,\ Y \mapsto 3]$.

Figure 2 shows via an example that the analysis is not distributive.

2.4 Effect systems and effect algebras

Type-and-effect systems extend type systems to analyse impure concepts such as IO, exceptions and mutable state [5, 7, 24, 15]. Type-and-effect judgements are often written as $\Gamma \vdash e : \tau \,\&\, F$ for an expression e of type τ in context Γ with potential effects described by F. (For the remainder of Section 2 we take e as ranging over general programming-language expressions.) For example, F might be the set of exceptions the expression e may throw. Type-and-effect systems also introduce *latent effect* annotations in functions, for example $\tau_1 \xrightarrow{F} \tau_2$ is the type of a function which has effect F when applied.

The simplest effect systems use powersets of symbols representing possible impure program actions, ignoring control-flow and statement order by using $\cup$ to combine effect information (e.g., [5, 24]). Consider the effect system that captures the set of exceptions that an expression may raise. In this case, the inference rule for conditionals is:

$$(\text{IF})\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \,\&\, F_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, F_2 \qquad \Gamma \vdash e_3 : \tau \,\&\, F_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \,\&\, F_1 \cup F_2 \cup F_3}$$

However, this only covers 'may' analyses, and not 'must' analyses, and furthermore it only allows for commutative effect combination. Amtoft et al. [1] therefore introduced separate operators for sequencing ($\rhd$) and combining alternate effects ($\sqcup$) e.g., in if-then-else-style conditionals. The meaning of $\rhd$ is such that $F_1 \rhd F_2$ is the cumulative effect of two sequenced operations, where the first has the effect F_1 and the second F_2. This sequential composition of effects, in a space D, is generally modelled as a monoid $(D, \rhd, 1)$ where $\rhd$ is an associative operation with identity element 1. The previous inference rule now becomes:

$$(\text{IF})\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \,\&\, F_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, F_2 \qquad \Gamma \vdash e_3 : \tau \,\&\, F_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \,\&\, F_1 \rhd (F_2 \sqcup F_3)}$$

The effect system now distinguishes sequencing from branching and allows the former to be non-commutative. Such effect systems, which take control flow into account, are sometimes referred to as *sequential* (or *flow-sensitive*) *effect systems* [26].

One algebraic characterisation of these effect-system operators, due to Katsumata [9], is as a partially-ordered[2] monoid (*pomonoid*), which we write as a quadruple $(D, \sqsubseteq, \rhd, 1)$ where D is both a poset (ordered by $\sqsubseteq$) and a monoid (with $\sqsubseteq$-monotonic operation $\rhd$).

Gordon argues for a special case of this model[3] called *effect quantales* [6]. An effect quantale $(D, \sqcup, \rhd)$ is a bounded join-semilattice where $\rhd$ distributes over $\sqcup$ on both sides: $x \rhd (y \sqcup z) = (x \rhd y) \sqcup (x \rhd z)$ and $(y \sqcup z) \rhd x = (y \rhd x) \sqcup (z \rhd x)$. Gordon adds the requirement that $(D, \sqcup)$ has a $\top$ element (but this holds whenever D is finite-height) and also that $\top$ is a left- and right-zero for $\rhd$.

Distributivity of $\rhd$ over $\sqcup$ implies its monotonicity w.r.t. $\sqsubseteq$, but not vice versa.

Both Katsumata's pomonoids and Gordon's effect quantales form bases for sequential effect systems. Effect quantales are a special case of pomonoids, but have the laudable aim to be closer to modelling only those effect algebras which are useful in practice. We however argue that the distributivity requirement of effect quantales is too strong (Section 3.4).

---

[2] Katsumata proposed a *pre-ordered monoid*, but this becomes a partially ordered monoid after quotienting by equivalence classes hence our slight re-characterisation here to match the partial-order setting of data-flow analysis.

[3] Here we consider only sequential composition and alternation; Gordon's work also considers iteration.

$$(\text{IF})\ \frac{\Phi(\ell) = \langle\!\langle v \rangle\!\rangle_{\text{TF}} \rhd (\Phi(\ell_1) \sqcup \Phi(\ell_2))}{\Phi \vdash (\ell : \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell_1\ \mathsf{else\ goto}\ \ell_2) : \mathsf{Int} \,\&\, \Phi(\ell)}$$

$$(\text{ASSIGN})\ \frac{\Phi(\ell) = \langle\!\langle \ell : X := e \rangle\!\rangle_{\text{TF}} \rhd \Phi(\ell')}{\Phi \vdash (\ell : X := e;\ \mathsf{goto}\ \ell') : \mathsf{Int} \,\&\, \Phi(\ell)} \qquad (\text{HALT})\ \frac{\Phi(\ell) = \langle\!\langle v \rangle\!\rangle_{\text{TF}}}{\Phi \vdash (\ell : \mathsf{halt}\ v) : \mathsf{Int} \,\&\, \Phi(\ell)}$$

Figure 3 Data-flow effect system for the imperative language of CFGs

3 An effect system for data-flow analysis

As discussed in Section 1, specific data-flow analyses have sometimes been given *ad hoc* characterisations as effect-system-like analyses (e.g., Nielson et al.'s annotated type system for reaching definitions [17]). Here we introduce a more general, unifying approach based on a type-and-effect system for CFGs in which statements in the language of Section 2.2 are given effect annotations corresponding to transfer functions. Since assignments and branches contain goto ℓ, their overall ('run to completion') effect does not directly correspond to traditional transfer functions of CFG nodes. Section 3.1 explores the details, introducing an effect system for liveness. Section 3.2 considers inference. Section 3.3 then generalises the system to classical dataflow analyses and constant propagation, which is non-distributive.

3.1 Type-and-effect system and inference rules for liveness

Recall the language of CFGs introduced in Section 2.2. We wrote $(\ell : S)$ to mean node ℓ is associated with statement S. We introduce judgements capturing the type and effect of running to completion a CFG program starting at a given statement. The judgement form is $\Phi \vdash (\ell : S) : \tau \,\&\, \phi$, where τ is a type and ϕ is an effect annotation given by a transfer function that is a *combination* of transfer functions on paths from ℓ up to a halt. More precisely: for liveness, applying ϕ to the live variable set at program exit (the boundary information) gives the live set at ℓ.

The role of Φ (which is a map from labels to transfer functions) is more subtle. Normally, type-and-effect systems are given in a syntax-directed manner. But loops in programs behave like recursive functions, requiring finding a fixed point (with potentially multiple solutions). Here, we posit a solution Φ giving the data-flow value at each program point, and use inference rules to assert this is consistent. There may be multiple possible Φ (fixed points).

The type-and-effect system of this form that describes liveness is given in Figure 3. It uses various functions and symbols. The operators $\sqcup$ and $\rhd$ are $\cup$ and $\circ$ (function composition), respectively. The notation $\langle\!\langle \ell : X := e \rangle\!\rangle_{\text{TF}}$ and $\langle\!\langle v \rangle\!\rangle_{\text{TF}}$ represents transfer functions corresponding respectively to assignments $X{:=}e$ at label ℓ, and variable references in halt v and if $v \geq 0$ statements. They are respectively $\lambda s.\ (s \setminus \mathit{kill}(\ell : X := e)) \cup \mathit{gen}(\ell : X := e)$ and $\lambda s.\ s \cup \mathit{fv}(v)$. We interpret these rules inductively and we are interested in the *least* solution (in terms of Φ in the partial order of functions from labels to transfer functions).

▶ **Theorem 1.** *Let $\hat{\Phi}$ be the least solution for a CFG that contains an instruction with label ℓ. Then $\hat{\Phi}(\ell)(\emptyset)$ is equal to the set of live variables at node ℓ of the CFG.*

Proof. This is restating a well-known fact about transfer functions by Sharir and Pnueli [22]. A statement and proof of it can be found in, for example, Theorem 7-3.4 in Muchnick and Jones [14]. Using their notation, the expression $\hat{\Phi}(\ell)(\emptyset)$ corresponds to $z_\ell = \chi_\ell(\emptyset)$ and the live set at ℓ is x_ℓ, and the theorem states that $x_\ell = z_\ell$. See Appendix A for details. ◀

Effect systems are traditionally applied to functional languages to analyse impure code. Thus calling our approach here an 'effect system' may seem unorthodox. Our justification is that the inference system in Figure 3 is the pre-image of the translation in Section 4 (based on the McCarthy transformation) from CFGs into functional code with a type-and-effect system (via graded monads) mapping transfer functions to type-based effect information.

Our effect system here resembles Nielson et al.'s "annotated type system" [17] capturing reaching-definitions analysis for a simple imperative `while` language. The main difference is that we operate with transfer functions on CFGs (which, to the best of our knowledge, is a novel approach), unifying several monotone data-flow analyses (shown in Section 3.3).

3.2 Inferring effects

Given a labelled imperative program as in Section 3.1, we want to find the effects associated with every single label. We present a method to infer the *principal* solution to this problem.

Statements in a CFG are uniquely labelled. Thus we can see a CFG as a set of tuples $(\ell : S)$, where ℓ is a label and S is a statement. Let ϕ_ℓ be the effect associated with the label ℓ, so that $\Phi \vdash (\ell : S) : \tau \,\&\, \phi_\ell$ holds. For every statement there is an associated set of constraints involving its effect. These constraints resemble the rules given in Figure 3. They are given in the form of inequalities that use a subeffecting relation $\sqsubseteq$, which in the case of liveness is just $\subseteq$ lifted to the function space. Each statement form below emits the indicated constraint (these are conventionally expressed using $\sqsupseteq$, the converse of $\sqsubseteq$):

$$
\begin{array}{lcl}
(\ell : \mathsf{halt}\ v) & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \\
(\ell : \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell_1\ \mathsf{else\ goto}\ \ell_2) & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi_{\ell_1} \sqcup \phi_{\ell_2}) \\
(\ell : X := e;\ \mathsf{goto}\ \ell') & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} \rhd \phi_{\ell'}
\end{array}
$$

We seek the least solution (w.r.t. $\sqsubseteq$) for this set of constraints. Since the domain of the constraints is the lattice of transfer functions, finding the least solution is done by using a simple work-list algorithm: Initially all ϕ_ℓ are set to $\bot_{DFValues \to DFValues}$ (the transfer function that maps any set of data-flow values to $\emptyset$). Then the solution is iteratively improved until we reach a tuple of transfer functions that satisfies all the constraints.

Since every transfer function appears on the left-hand side of exactly one constraint, the value in the next iteration is updated according to this constraint. For example, if there is a constraint $\phi \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi_1 \sqcup \phi_2)$, this update step sets the new estimate of ϕ to exactly $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi'_1 \sqcup \phi'_2)$, where ϕ'_1 and ϕ'_2 are the current estimates of ϕ_1 and ϕ_2. These steps are monotonic with respect to $\sqsubseteq$, and thus this iteration converges to the least fixed-point solution for our finite-height lattices.

3.3 Generalising to other data-flow analyses

Our CFG-based effect system for liveness can be generalised to a single framework capturing the four classical data-flow analyses (live variables, reaching definitions, very busy expressions, available expressions). The generalised form is parameterised by the following algebra:

- a set of data-flow values *DFValues*, effects are then transfer functions drawn from $DFValues \to DFValues$;
- a subeffecting relation $\sqsubseteq$ on transfer functions;
- a sequencing operator $\rhd$ on transfer functions;
- a function $\langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}$ mapping labelled assignment statements to transfer functions;
- a function $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}}$ mapping a potential variable appearing in $\mathsf{halt}\ v$ or in $\mathsf{if}\ v \geq 0$ to a transfer function representing its being read.

	$DFValues$	$\sqsubseteq$	$\sqcup$	$\rhd$	$kill(\ell : X := e)$	$gen(\ell : X := e)$	$vgen(v)$
LVA	$\mathcal{P}(Vars)$	$\subseteq$	$\cup$	$\circ$	$\{X\}$	$fv(e)$	$fv(v)$
RD	$\mathcal{P}(Vars \times Labels)$	$\subseteq$	$\cup$	$\hat{\circ}$	$\{(X, l) \mid l \in Labels\}$	$\{(X, \ell)\}$	$\emptyset$
VBE	$\mathcal{P}(Expressions)$	$\supseteq$	$\cap$	$\circ$	$\{e' \mid X \in fv(e')\}$	$\{e\}$	$fv(v)$
AVAIL	$\mathcal{P}(Expressions)$	$\supseteq$	$\cap$	$\hat{\circ}$	$\{e' \mid X \in fv(e')\}$	$\{e\}$	$fv(v)$

(Recall $v ::= X \mid k$ therefore $fv(v)$ in the rightmost column is either a singleton or empty set.)

Figure 4 CFG effect-system instantiations for classical data-flow analyses

Sets of transfer functions equipped with $\sqsubseteq$ are lattices, therefore a $\sqcup$ operator (join) exists. For all four classical analysis, $\langle\!\langle \ell : X := e \rangle\!\rangle_{\text{TF}}$ and $\langle\!\langle v \rangle\!\rangle_{\text{TF}}$ can be expressed:

$$\begin{aligned} \langle\!\langle \ell : X := e \rangle\!\rangle_{\text{TF}} &= (\lambda d.\ (d \setminus kill(\ell : X := e)) \cup gen(\ell : X := e)) &&: DFValues \to DFValues \\ \langle\!\langle v \rangle\!\rangle_{\text{TF}} &= (\lambda d.\ d \cup vgen(v)) &&: DFValues \to DFValues \end{aligned}$$

The space of data-flow values $DFValues$ along with its $\sqsubseteq$, $\sqcup$, gen, $kill$ and $vgen$ operators are variously parameterised for the four data-flow analyses as shown in Figure 4. The effect system that then describes all of these is precisely the one given in Figure 3.

In these instantiations, the $\rhd$ operator is particularly interesting. The algebra $(DFValues \to DFValues, \sqsubseteq, \rhd, id)$ is a partially ordered monoid, with id as the unit element. We consider two possibilities for $\rhd$ depending on the direction of the analysis

- For backwards analysis, $\rhd$ is function composition $\circ$;
- For forwards analysis, $\rhd$ is reverse function composition $\hat{\circ}$ – defined as $f \mathbin{\hat{\circ}} g \stackrel{\text{def}}{=} g \circ f$.

3.4 Constant propagation as a non-distributive example

Constant propagation from Section 2.3 (not one of the four classical analyses) also fits into the above framework. We take $DFValues$ to be the lattice of mappings s from variables to $\mathbb{Z}^{\top}_{\bot}$ with the $\sqsubseteq$ relation being lifted component-wise. This lattice of transfer functions becomes a pomonoid by taking $\rhd$ to be reverse composition $\hat{\circ}$ (since constant propagation is a forwards analysis). Transfer functions for assignment and variable access are:

$$\langle\!\langle \ell : X := e \rangle\!\rangle_{\text{TF}} = \lambda s.\ s[X \mapsto s(e)] \qquad \text{and} \qquad \langle\!\langle v \rangle\!\rangle_{\text{TF}} = \lambda s.\ s$$

where we abusively write $s(e)$ to mean the value in $\mathbb{Z}^{\top}_{\bot}$ obtained by substituting variables in e as specified by s and simplifying. Variable access does not update variables so $\langle\!\langle v \rangle\!\rangle_{\text{TF}} = id$.

As an example, sequencing the effects of $X := 1$ and $Y := X + 2$ gives the effect:

$$\begin{aligned} (\lambda s.\ s[X \mapsto 1]) \mathbin{\hat{\circ}} (\lambda s.\ s[Y \mapsto s(X) + 2]) &= (\lambda s.\ s[Y \mapsto s(X) + 2]) \circ (\lambda s.\ s[X \mapsto 1]) \\ &= \lambda s.\ s[X \mapsto 1, Y \mapsto 3] \end{aligned}$$

Thus the inference system of Figure 3 can be used also for constant propagation.

With constant propagation, $\sqcup$ and $\hat{\circ}$ do not satisfy distributivity, as seen previously in the example of Figure 2. In this algebra, that example illustrates the fact that for:

$$\phi_1 = \lambda s.\ s[X \mapsto 1, Y \mapsto 2] \qquad \phi_2 = \lambda s.\ s[X \mapsto 2, Y \mapsto 1] \qquad \phi_3 = \lambda s.\ s[Z \mapsto s(X) + s(Y)]$$

distributivity is violated – that is, $(\phi_1 \mathbin{\hat{\circ}} \phi_3) \sqcup (\phi_2 \mathbin{\hat{\circ}} \phi_3) \neq (\phi_1 \sqcup \phi_2) \mathbin{\hat{\circ}} \phi_3$. Thus, the idea of basing effect-systems on the distributive structure of quantales (as in [6]) would exclude this common static analysis. We therefore advocate that distributivity is not imposed (Section 5).

4 Translating to a graded monadic setting

We now formulate a graded monadic model of the effect system given in Section 3, exploring the use of graded structures to encode liveness analysis in programming and semantic modelling. We describe translations from our CFGs into a pure functional language (e.g., Haskell, Agda, Coq, or a pure subset of ML).

We briefly overview graded monads in Section 4.1. We go on to define a monadic variant of the McCarthy transformation in Section 4.2; this is generalised to a graded monadic McCarthy transformation in Section 4.3.1. For the graded monadic case, the data-flow equations get represented as typing constraints in the target language's type system. Section 4.3.2 gives a graded monad which further refines a semantic model of state by liveness information. Section 4.4 considers a concrete translation into Haskell, details of which are in Appendix B. Lastly, Section 4.5 explains how to generalise this approach to other data-flow analyses.

4.1 Graded monads

Monads are common in pure functional programming languages (such as Haskell) for embedding and structuring effectful computations [27] and for semantic models of effects [13]. We recall a programming oriented definition: a monad is a triple $(M, \gg\!=, \mathsf{return})$ where M is a type constructor, $\gg\!=$ (bind) is an infix operator, and return is a function, with the types:

$$\mathsf{return} : \forall\alpha.\ \alpha \to M\alpha \qquad (\gg\!=) : \forall\alpha\forall\beta.\ M\alpha \to (\alpha \to M\beta) \to M\beta$$

Following Moggi [13], we use the word *computation* for values of type $M\tau$, just like we use *function* for values of type $\sigma \to \tau$.

In addition, these operations should satisfy the following axioms:

$$m \gg\!= \mathsf{return} = m \qquad \text{(right identity)}$$
$$\mathsf{return}\ x \gg\!= f = f\ x \qquad \text{(left identity)}$$
$$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x.\ f\ x \gg\!= g) \qquad \text{(associativity)}$$

Wadler and Thiemann [28] showed that monads and effect systems can be united by transposing effect systems into an equivalent monadic system with effect annotations in types: for an expression of type τ with effect F, there is an monad M annotated with F (written M_F) such that there is an equivalent expression of type $M_F\tau$. This annotated monad represents the possible effects of an impure expression e, described by F. *Graded monads* essentially systematise and generalise this idea so that a model or embedding of the effectful computation may depend on the effect information F, which has some algebraic structure (the *effect algebra*). In this way, graded monads can capture effect information in types (e.g., for fine-grained effect and resource reasoning) or make effect semantics more fine-grained.

Graded monads generalise monads to an indexed family of type constructors whose indices range over elements of a given algebraic structure [4, 9, 15]. The operations of this structure then mediate the operations of the graded monad. The structure of grades is usually a pomonoid $(D, \sqsubseteq, \triangleright, 1)$, giving a graded monad $(\{M^r\}_{r\in D}, \mathsf{sub}, \gg\!=, \mathsf{return})$, where $\{M^r\}_{r\in D}$ is a family of type constructors indexed by D-elements and $\gg\!=$, return, and sub have types:

$$\begin{aligned}
\gg\!= &: \forall r\forall s\forall\alpha\forall\beta.\ M^r\alpha \to (\alpha \to M^s\beta) \to M^{r\triangleright s}\beta \\
\mathsf{return} &: \forall\alpha.\ \alpha \to M^1\alpha \\
\mathsf{sub} &: \forall r\forall s\forall\alpha.\ M^r\alpha \to M^s\alpha \qquad \textit{if } r \sqsubseteq s
\end{aligned}$$

Here $\gg\!=$ and sub are polymorphic in types and grades. We use Greek letters for types and Roman letters for elements (grades) of the algebra in order to avoid clutter in type signatures.

A graded monad satisfies axioms analogous to those of a monad, but with the addition of grades in such a way that the graded monad laws depend on the associativity and identity properties of the monoid, where for all $x : \alpha, m : M^r\alpha, f : \alpha \to M^s\beta$ and $g : \beta \to M^t\gamma$:

$$\begin{aligned} m \gg\!=_{r,1} \mathsf{return} &= m && : M^r\alpha && \text{(right identity)} \\ \mathsf{return}\ x \gg\!=_{1,s} f &= f\ x && : M^s\beta && \text{(left identity)} \\ (m \gg\!=_{r,s} f) \gg\!=_{r \triangleright s,t} g &= m \gg\!=_{r,s \triangleright t} (\lambda x.\ f\ x \gg\!=_{s,t} g) && : M^{r \triangleright s \triangleright t}\gamma && \text{(associativity)} \end{aligned}$$

We subscript the operations with the instantiation of the grades here for clarity.

The sub satisfies the following $\forall r, s, r', s', m : M^r\alpha, f : \alpha \to M^s\beta$ where $r \sqsubseteq r'$ and $s \sqsubseteq s'$:

$$\mathsf{sub}_{r,r'} m \gg\!=_{r',s'} (\mathsf{sub}_{s,s'} \circ f) = \mathsf{sub}_{r \triangleright s, r' \triangleright s'} (m \gg\!=_{r,s} f) : M^{r' \triangleright s'}\beta \qquad \text{(monotonicity)}$$

Categorically, graded monads correspond to lax monoidal functors between a pomonoid (viewed as a category) and a category of endofunctors (essentially type constructors) [9, 20]. This categorical construction embodies the idea that graded monads match the structure of some analysis domain (a pomonoid on D) to the structure of a semantic domain (type constructors modelling computations). The resulting operations ($\gg\!=$, return, sub) propagate the pomonoid structure with them via the grades, describing the structure of a computation.

4.2 Monadic McCarthy transformation

McCarthy's transformation [12] maps CFG statements to mutually recursive function definitions using an m-tuple of functionally updated variables to represent the state. For example, the node $(\ell_1 : Y := X + Z;\ \mathsf{goto}\ \ell_2)$ in a CFG containing variables X, Y, Z can be translated into the function $f_1(x, y, z) = f_2(x, x + z, z)$ where f_2 is the function corresponding to the CFG node with label ℓ_2. We define a variant, using a monad to represent state, and call it the *monadic McCarthy transformation*.

The standard state monad [27] models a single mutable memory cell with type constructor State α parameterised by the type of values that can be stored α, and two operations for manipulating the state: get : $\forall\alpha.$State α α and put : $\forall\alpha.\alpha \to$ State α Unit. We can thus represent m integer variables by the monad State(Int, ..., Int) with access to each variable provided by get and put and projections. An alternative is to use a monad transformer stack. For brevity, we instead assume an equivalent monad MultiState which holds the state of m integer variables and has m operations get_i : MultiState Int and put_i : Int $\to$ MultiState Unit one for each CFG variable ($X, Y, Z, \ldots \in$ *Vars*). In examples these are written getX, putY etc. Thus MultiState τ is the type of computations over mutable variables that return type τ.

We use Haskell's do $\{\ldots\}$ notation as syntactic sugar for monadic computations,[4] equivalent to Moggi's monadic metalanguage [13]. For example, do $\{x \leftarrow e_1;\ e_2\}$ sugars $e_1 \gg\!= (\lambda x.\ e_2)$, and do $\{e_1;\ e_2\}$ sugars $e_1 \gg\!= (\lambda\_.\ e_2)$. The desugaring is recursively applied.

Our monadic McCarthy transformation produces a set of mutually recursive definitions of computation values of monadic type (in our case MultiState Int) instead of a set of mutually recursive functions. Each labelled statement $(\ell : S_\ell)$ maps to a definition $[\![\ell : S_\ell]\!]_{\mathrm{MM}}$ as specified in Figure 5: for assignment $X := e$, the variables of e are read into temporary (pure) variables using get, followed by a putX to write to X; variables read within conditionals and halts are treated similarly. The resultant monadic definitions can be directly read as a Haskell program (or an ML program after desugaring into recursive function definitions).

[4] There is an additional assumption of monad *strength* which allows monadic computations to close over variables in scope. Strength holds for all monads in Cartesian-closed categories (and in programming settings). The notion of strength extends to graded monads [9] and is provided for all our examples.

$\ell : S$	$[\![\ell : S]\!]_{\mathrm{MM}}$
$\ell : X := Y + Z;\ \mathsf{goto}\ \ell'$	$g_\ell = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ z \leftarrow \mathsf{getZ};\ \mathsf{putX}\ (y+z);\ g_{\ell'}\ \}$
$\ell : X := k;\ \mathsf{goto}\ \ell'$	$g_\ell = \mathsf{do}\ \{\ \mathsf{putX}\ k;\ g_{\ell'}\ \}$
$\ell : \mathsf{if}\ X \geq 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else\ goto}\ \ell''$	$g_\ell = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{if}\ x \geq 0\ \mathsf{then}\ g_{\ell'}\ \mathsf{else}\ g_{\ell''}\ \}$
$\ell : \mathsf{halt}\ X$	$g_\ell = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{return}\ x\ \}$

Figure 5 The monadic McCarthy transformation $[\![-]\!]_{\mathrm{MM}}$. For assignment, we only give the cases $X := k$ and $X := Y + Z$; other cases, e.g., $X := Y + 1$, are similar. Conditional and halt forms that have k (constants) instead of variables X are analogous.

$$
\begin{array}{l}
\ell_0 : X := 100;\ \mathsf{goto}\ \ell_1 \\
\ell_1 : \mathsf{if}\ X \geq 0\ \mathsf{then\ goto}\ \ell_2\ \mathsf{else\ goto}\ \ell_4 \\
\ell_2 : X := X - 1;\ \mathsf{goto}\ \ell_3 \\
\ell_3 : Y := Y + 1;\ \mathsf{goto}\ \ell_1 \\
\ell_4 : R := Y + Z;\ \mathsf{goto}\ \ell_5 \\
\ell_5 : \mathsf{halt}\ R
\end{array}
\xrightarrow{[\![-]\!]_{\mathrm{MM}}}
\begin{array}{l}
g_0 = \mathsf{do}\ \{\ \mathsf{putX}\ 100;\ g_1\ \} \\
g_1 = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX}; \mathsf{if}\ x \geq 0\ \mathsf{then}\ g_2\ \mathsf{else}\ g_4\ \} \\
g_2 = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{putX}\ (x-1);\ g_3\ \} \\
g_3 = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ \mathsf{putY}\ (y+1);\ g_1\ \} \\
g_4 = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ z \leftarrow \mathsf{getZ};\ \mathsf{putR}\ (y+z);\ g_5\ \} \\
g_5 = \mathsf{do}\ \{\ r \leftarrow \mathsf{getR};\ \mathsf{return}\ r\ \}
\end{array}
$$

Figure 6 Example monadic McCarthy transformation. CFG code (left) is translated into mutually recursive definitions of computation values (right).

Figure 6 exemplifies the monadic McCarthy transformation converting an imperative program (left) to a set of mutually recursive computation definitions (right).

The monadic McCarthy transformation produces a program with equivalent behaviour to the original CFG (by a straightforward refactoring of McCarthy's transformation into the state monad). Next, we show that a more refined model can be given by targeting a *graded monad* instead of a monad. This allows the target of the translation to capture the same data-flow information as the CFG effect system's judgements.

4.3 Graded monadic McCarthy transformation for liveness

The above monadic McCarthy transformation maps CFG terms to state monad computations, i.e., $[\![l : S]\!]_{\mathrm{MM}} : \mathsf{MultiState}\ \mathsf{Int}$. Instead, given a graded monad $\mathsf{MultiState}^{\phi}$ which provides state monad-like behaviour (graded by our pomonoid of transfer functions ϕ), we give a *graded monadic* McCarthy transformation $[\![\ell : S]\!]_{\mathrm{GM}} : \mathsf{MultiState}^{\Phi(\ell)}\ \tau$ whenever $\Phi \vdash (\ell : S) : \tau \,\&\, \Phi(\ell)$.

We describe this graded monadic McCarthy transformation (Section 4.3.1) by first taking the usual MultiState monad and wrapping into a trivial graded monad: one whose grades only decorate the types but do not affect the operations and thus have no semantic meaning. We then replace this graded monad with one whose grades have semantic meaning, refining the types and operations of the former to give a semantic account of liveness (Section 4.3.2).

4.3.1 Transformation to a trivial graded monad

Given a monad M and a pomonoid $(D, \sqsubseteq, \triangleright, 1)$ one can construct a *trivial* graded monad with type constructors $M^d_{\mathsf{triv}}\tau = M\tau$ for $d \in D$. In this construction M^d_{triv} simply wraps M and thus the grades have no bearing on the computation encoded by the monad. The monad operations of M provide the graded monad operations of M_{triv} via this wrapping, and the required graded monad laws follow from the laws of the monoid $(D, \sqsubseteq, \triangleright, 1)$ and monad M.

We use this construction on the MultiState monad to form a graded monad written $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}$ graded by the pomonoid of transfer functions $(DFValues \to DFValues, \sqsubseteq, \rhd, id)$ from Section 3.1. Thus a value of type $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \tau$ is a stateful computation that returns a value of type τ with some transfer-function grade ϕ associated to it by its operations.

The graded monadic McCarthy transformation $[\![-]\!]_{\mathrm{GM}}$ enriches $[\![-]\!]_{\mathrm{MM}}$ (Fig. 5) in only two ways: (i) applying sub to both $g_{\ell'}$ and $g_{\ell''}$ in the translation of if, and (ii) using the *graded* monad operations below in the body of do (and for $\gg\!=$ when desugaring it):

$$\begin{array}{lcl} \mathsf{getX} & : & \mathsf{MultiState}^{gen_X}_{\mathsf{triv}}\ \mathsf{Int} \\ \mathsf{putX} & : & \mathsf{Int} \to \mathsf{MultiState}^{kill_X}_{\mathsf{triv}}\ \mathsf{Unit} \\ \mathsf{return} & : & \forall\alpha.\alpha \to \mathsf{MultiState}^{id}_{\mathsf{triv}}\ \alpha \\ \gg\!= & : & \forall\phi,\phi',\alpha,\beta.\ \mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \alpha \to (\alpha \to \mathsf{MultiState}^{\phi'}_{\mathsf{triv}}\ \beta) \to \mathsf{MultiState}^{\phi \rhd \phi'}_{\mathsf{triv}}\ \beta \end{array} \quad \text{where} \quad \begin{array}{lcl} gen_X & \stackrel{\text{def}}{=} & \lambda d.\ d \cup \{X\} \\ kill_X & \stackrel{\text{def}}{=} & \lambda d.\ d \setminus \{X\} \end{array}$$

Since the transformation operates on the syntax of CFGs, rather than judgements of the CFG effect system, the grades on each computation type must be inferred by generating a set of typing constraints which are then solved (as was done in Section 3.2), by the host language's type system (we consider the feasibility of this in Section 4.4).

The syntactic translation results in graded monadic computations whose grades match exactly the analysis of our CFG effect-system from Section 3:

▶ **Lemma 2** (Soundness of the graded monadic McCarthy transformation). *If* $\Phi \vdash (\ell : S) : \tau \& \Phi(\ell)$ *and* $\forall \ell' \in \mathsf{dom}(\Phi).(g_{\ell'} : \mathsf{MultiState}^{\Phi(\ell')}_{\mathsf{triv}}\ \mathsf{Int})$ *then* $[\![\ell : S]\!]_{GM} : \mathsf{MultiState}^{\Phi(\ell)}_{\mathsf{triv}}\ \tau$.

▶ **Example 3.** Let g, of type $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \mathsf{Int}$, represent a liveness transfer function 'for the rest of the program'. Now consider the following expression (effectively prefixing g with the statement $Z := X + Y$ and applying the graded McCarthy transformation $[\![-]\!]_{\mathrm{GM}}$ above):

$$\mathsf{do}\ \{\quad x \leftarrow \mathsf{getX};\ y \leftarrow \mathsf{getY};\ \mathsf{putZ}\ (x + y);\ g\quad \}$$

By construction, its type is $\mathsf{MultiState}^{\phi'}_{\mathsf{triv}}\ \mathsf{Int}$ where $\phi' = \lambda d.\ (\phi(d) \setminus \{Z\}) \cup \{X, Y\}$ represents the liveness transfer function for $Z := X + Y$ followed by the 'rest of the program' because

$$\phi' = gen_X \rhd gen_Y \rhd kill_Z \rhd \phi = \lambda d.\ (\phi(d) \setminus \{Z\}) \cup \{X, Y\}$$

As in Section 3.1, $\phi'(\emptyset)$ gives us the liveness information the start of the 'body' of do $\{\}$ because the boundary information is that the set of live variables is empty at program exit.

▶ **Example 4.** In Figure 6, we converted an imperative program into mutually recursive computation values $g_0, \ldots, g_5$. Let $\phi_0, \ldots, \phi_5$ stand for the transfer function grades of the graded monadic types of $g_0, \ldots, g_5$, so that g_i is of type $\mathsf{MultiState}^{\phi_i}_{\mathsf{triv}}\ \mathsf{Int}$. Then these transfer functions must satisfy the following constraints (coming from the type system):

$$\begin{array}{lcl} \phi_0 & \sqsupseteq & kill_X \rhd \phi_1 \\ \phi_2 & \sqsupseteq & gen_X \rhd kill_X \rhd \phi_3 \\ \phi_4 & \sqsupseteq & gen_Y \rhd gen_Z \rhd kill_R \rhd \phi_5 \end{array} \qquad \begin{array}{lcl} \phi_1 & \sqsupseteq & gen_X \rhd \phi_2 \\ \phi_3 & \sqsupseteq & gen_Y \rhd kill_Y \rhd \phi_1 \\ \phi_5 & \sqsupseteq & gen_R \rhd id \end{array} \qquad \begin{array}{lcl} \phi_1 & \sqsupseteq & gen_X \rhd \phi_4 \end{array}$$

The usual fixed-point iteration gives the principal (least) solution: $\phi_0 = \lambda d.(d \setminus \{X, R\}) \cup \{Y, Z\}$, $\phi_1 = \phi_2 = \phi_3 = \phi_4 = \lambda d.\ (d \setminus \{R\}) \cup \{X, Y, Z\}$ and $\phi_5 = \lambda d.\ d \cup \{R\}$. The set of live variables at the program start (i.e. at g_0) is therefore $\phi_0(\emptyset) = \{Y, Z\}$.

4.3.2 Analysis-directed semantics via grade-based refinement

The previous section constructed the graded monad $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\,\tau$ as a simple wrapper over the usual state monad; grade ϕ was a transfer function but it had no semantic meaning: the grades were merely decorations on types and did not affect the operations. We can instead use grades to *refine* the types and operations of the usual state monad by the liveness information, so that graded monad operations actually depend on the grades. In this case, refinement means restricting stores to subsets of the variables involved in a program. This paves the way to ensuring that only semantically valid analyses can be encoded as grades. The translation $[\![\ell : S]\!]_{\mathrm{GM}}$ remains the same but we instead replace the operations of $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}$ with the operations of a new graded monad $\mathsf{MultiState}^{\phi}$ (such that Lemma 2 holds for $\mathsf{MultiState}^{\phi}$).

Previously, MultiState and $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\,\tau$ represented their stores as m-tuples of Ints where $m = |\mathit{Vars}|$ (the CFG variables), i.e. $\mathsf{MultiState}\,\tau = (\mathsf{Int},\ldots,\mathsf{Int}) \to \tau \times (\mathsf{Int},\ldots,\mathsf{Int})$. Now, given any subset $V \subseteq \mathit{Vars}$, we define the *V-refined store* $\mathsf{Store}(V)$ to be $V \to \mathsf{Int}$, writing $\hat{\emptyset}$ for the only member of $\mathsf{Store}(\emptyset)$. Note, $\mathsf{Store}(\mathit{Vars})$ recovers (up to isomorphism) the previous $(\mathsf{Int},\ldots,\mathsf{Int})$.

We now define $\mathsf{MultiState}^{\phi}$ whose input and output stores are computed from ϕ:

$$\mathsf{MultiState}^{\phi}\tau = \mathsf{Store}(\mathit{reads}(\phi)) \to \tau \times \mathsf{Store}(\mathit{footprint}(\phi))$$

For the input store, $\mathit{reads}(\phi) = \phi(\emptyset)$ gives us the subset of variables which are live-in and thus read by a computation of type $\mathsf{MultiState}^{\phi}\tau$. For the output store, $\mathit{footprint}(\phi) = \phi(\emptyset) \cup (\mathit{Vars} \setminus \phi(\mathit{Vars}))$ gives the *footprint* (borrowing terminology from separation logic [18]) containing those variables read or written by this computation.[5] For example, do $\{x \leftarrow \mathsf{getX};\ y \leftarrow \mathsf{getY};\ \mathsf{putZ}\ (x+y);\}$ has grade $\phi = \mathit{gen}_X \rhd \mathit{gen}_Y \rhd \mathit{kill}_Z = \lambda d.\ (d \setminus \{Z\}) \cup \{X,Y\}$ (akin to Example 3) and thus $\mathit{reads}(\phi) = \phi(\emptyset) = \{X,Y\}$ and $\mathit{footprint}(\phi) = \{X,Y,Z\}$.

The $\mathsf{MultiState}^{\phi}$ type is a graded monad with refined return and state operations:

$$\begin{array}{lll}
\mathsf{return} : \forall\alpha.\alpha \to \mathsf{MultiState}^{id}\alpha & = \lambda x.\lambda s.(x,\hat{\emptyset}) & : \forall\alpha.\alpha \to (\mathsf{Store}(\emptyset) \to \alpha \times \mathsf{Store}(\emptyset)) \\
\mathsf{getX} : \mathsf{MultiState}^{\mathit{gen}_X}\,\mathsf{Int} & = \lambda s.(s(X),s) & : \mathsf{Store}(\{X\}) \to \mathsf{Int} \times \mathsf{Store}(\{X\}) \\
\mathsf{putX} : \mathsf{Int} \to \mathsf{MultiState}^{\mathit{kill}_X}\,\mathsf{Unit} & = \lambda x.\lambda s.((),[X \mapsto x]) & : \mathsf{Int} \to (\mathsf{Store}(\emptyset) \to \mathsf{Unit} \times \mathsf{Store}(\{X\}))
\end{array}$$

On the right, we repeat the type of the operations, expanding the definition of $\mathsf{MultiState}^{\phi}$. The input and output stores of return are both the empty map as $\mathit{reads}(id) = \mathit{footprint}(id) = \emptyset$ representing that no variables are read or written by return. Thus, return represents a pure computation as isomorphic to the identity. The getX and putX operators are similarly refined.

The graded monad $\gg\!=$ resembles the usual state monad $\gg\!=$ but with three auxiliary operations ($\blacktriangleleft$, $\triangleleft$, and $\downarrow$ below) to manage the variously refined stores:

$$\begin{array}{rll}
\gg\!= & : \forall\phi,\phi',\alpha,\beta.\ \mathsf{MultiState}^{\phi}\alpha \to (\alpha \to \mathsf{MultiState}^{\phi'}\beta) \to \mathsf{MultiState}^{\phi \rhd \phi'}\beta & \\
& = \lambda m.\lambda f.\lambda s.\ \mathsf{let}\ (a,s') = m(\downarrow_{\phi,\phi'}\ s)\ \mathsf{in} & \\
& \qquad \mathsf{let}\ (b,s'') = (f\ a)(s \blacktriangleleft_{\phi,\phi'}\ s')\ \mathsf{in}\ (b,\ s' \triangleleft_{\phi,\phi'} s'') & \\
\textit{where}\ \downarrow_{\phi,\phi'} & : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \to \mathsf{Store}(\mathit{reads}(\phi)) & (\mathit{restrict}) \\
\blacktriangleleft_{\phi,\phi'} & : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \times \mathsf{Store}(\mathit{footprint}(\phi)) \to \mathsf{Store}(\mathit{reads}(\phi')) & (\mathit{merge}_1) \\
\triangleleft_{\phi,\phi'} & : \mathsf{Store}(\mathit{footprint}(\phi)) \times \mathsf{Store}(\mathit{footprint}(\phi')) \to \mathsf{Store}(\mathit{footprint}(\phi \rhd \phi')) & (\mathit{merge}_2)
\end{array}$$

Here $\downarrow_{\phi,\phi'}\ s$ restricts the incoming store $s : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi'))$ to $\mathsf{Store}(\mathit{reads}(\phi))$, i.e., just those variables live in computation $m : \mathsf{MultiState}^{\phi}\alpha$. The operation $s \blacktriangleleft_{\phi,\phi'}\ s'$ pads the

[5] A more refined graded state monad would return an output store containing only those variables that are written-to (e.g. as in [15]). However, liveness analysis alone does not allow us to compute just the variables written-to. The footprint is therefore a safe over-approximation of the written-to set.

domain of store $s' : \mathsf{Store}(\mathit{footprint}(\phi))$ (resulting from m) with the variables in store s, to produce a store whose domain is just the live variables required for computation $(f\ a)$. The $s' \lhd_{\phi,\phi'} s''$ operation similarly pads the domain of store $s'' : \mathsf{Store}(\mathit{footprint}(\phi'))$ (resulting from $(f\ a)$) with the variables in store s' to give the final updated store its required domain.

The resulting $\gg\!=$ operation thus 'filters' input stores by what is live, and output stores by the footprint enabling, e.g., soundness of dead-code removal to be proved (future work). The usual state monad $\gg\!=$ is recovered by redefining $\downarrow_{\phi,\phi'} s = s$ and $s \blacktriangleleft_{\phi,\phi'} s' = s \lhd_{\phi,\phi'} s' = s'$.

Appendix C provides more details and the proof that this is indeed a graded monad. For brevity, we omit the definition of sub.

4.4 Targeting a host language and applications

We have used Haskell-like do-notation as syntactic sugar for the (graded) monad operations in some functional language. We can take this a step further, concretely targeting GHC/Haskell, leveraging its combination of a practical functional language with an advanced type system. Appendix B gives more details, showing how the Section 4.3.1 can be captured in Haskell.

This approach works well for sequential code, but reaches its limits with branching and recursion as GHC does not have an appropriate notion of subtyping nor can it compute fixed-points of type equations. A system with subtyping and equirecursive types (e.g., OCaml) may fare better. An alternate approach is to make the graded monadic McCarthy transformation not just syntax directed but *type-and-effect directed*. In this approach, solutions to the data-flow equations can be computed (e.g, by work-list algorithm) *before* applying the graded monadic McCarthy transformation. The resulting (least) transfer functions can then be used in the translation to specialise the types of the resultant graded monadic program.

Whilst Haskell is shown as a target here, our approach is likely to be more useful in the setting of a proof assistant when formalising language semantics or a compiler and its optimisations (e.g., the CakeML verified compiler [25]).

4.5 Generalising to other data-flow analyses

So far we focused on liveness, where assignment statements are decomposed into sequences of get and put operations. For other data-flow analyses, we cannot perform the same translation of assignment as it may not be similarly decomposable. For example, for available expressions we cannot associate *kill* with put nor *gen* with get. To capture these other data-flow analyses, we can parameterise our graded monadic McCarthy transformation by a specialised interpretation for assignments $[\ell : x := e]_{\mathrm{GM}} : M^{\langle\!\langle \ell : x := e \rangle\!\rangle_{\mathrm{TF}}}\,\mathsf{Unit}$, graded by the assignment transfer function. The translation is then the same as Section 4.3, but with assignments translated as:

$$[\![\ell : x := e;\ \mathsf{goto}\ \ell']\!]_{\mathrm{GM}} = \mathsf{do}\,\{\,[\ell : x := e]_{\mathrm{GM}};\ g_{\ell'}\,\}$$

We then require that $[\ell : x := e]_{\mathrm{GM}}$ simulates the behaviour of assignment in the non-graded monadic McCarthy transformation $[\![-]\!]_{\mathrm{MM}}$ on the MultiState monad. Using this generalisation for different analyses with specialised graded monads akin to Section 4.3.2 is further work.

5 Conclusions and discussion

We demonstrated that a type-and-effect system based on transfer functions can be used to compute data-flow values at any point in a CFG, and in particular can be used for liveness analysis. Furthermore we have shown that the McCarthy transformation can be adapted

into a (graded) monadic form which embeds live variable analysis using control-flow graphs into functional programs, where transfer functions are grades of a graded monad.

This not only unifies two separately developed fields, but also contributes to the evolving discussion of "what properties do we expect of the effect algebras used as grades". In particular, it shows that the distributivity axiom posited for effect quantales is over-restrictive in that it does not allow representation of non-distributive data-flow problems such as constant propagation. Using a pomonoid (or even pre-ordered monoid as originally phrased by Katsumata) seems to impose *minimal requirements* on a model and so is most general. However, it then admits partial orders in which there is no concept of a least (or principal) solution and which do not seem to model any known static analysis. We suggest an appropriate model should be a pomonoid $(D, \sqsubseteq, \triangleright)$ which satisfies the following requirements:

- $\triangleright$ is monotonic w.r.t. $\sqsubseteq$ (following Katsumata and allowing distributivity);
- D is *bounded complete*: whenever set $X \subseteq D$ has some upper bound then it has a least upper bound.

An advantage of our graded-monadic approach compared to classical data-flow analysis is the potential for a *correct-by-construction property*; correctness of an analysis can be established at the semantic level, either denotationally or by showing a graded-type-preservation property in a reduction-style operational semantics. Correctness then follows from a number of results:

1. that live-variable analysis is achieved by fixed-point calculation over equations on transfer functions (Theorem 1);
2. soundness of the McCarthy transformation (established in [12]); soundness of replacing McCarthy's explicit state passing with the state monad (well-known); and soundness of our novel transformation to a graded state monad (Lemma 2);
3. that our graded monad $\mathsf{MultiState}^{\phi}$ really is a graded monad (Section 4.3.2 / Appendix C);
4. that reduction in our (graded monad) calculus exhibits progress and preservation.

The last point is the subject of future work, which we wish to explore in the context of general data-flow analyses and proving the correctness of program transformations.

Related work Benton et al. [2] use a graded-monad-based effect system to model non-determinism in an otherwise pure functional language and then use this information in a logical relation semantics to prove program transformations correct, whereas our focus is on embedding general data-flow analyses for imperative languages into graded monads.

Dijkstra monads [23] are a generalisation of monads used for verifying program conditions, where the annotation carries the precondition and postcondition of an expression. While more general, they achieve their full power in a dependently typed language. By contrast, we manage to get far in a graded monadic setting without dependent types.

Further work As discussed in Section 4.5, further work is to study the graded monadic McCarthy approach in more detail for analyses other than liveness, which was our focus here.

The current work embeds *intra-procedural* program analyses on control-flow graphs as grading inference problems in graded-monadic forms of effect systems. Further work might include showing how notions from inter-procedural analysis, such as context-sensitivity and the IDE and IFDS frameworks of Reps et al. [21], along with how notions such as bidirectional analysis fit into the 'properties as grades of a graded monad' model.

Section 4.4 discussed how the graded monadic embedding is likely to be most useful in the setting of verifying optimising compilers (rather than, say, general Haskell programming). Exploring our approach in this context is future work.

References

1 Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. *Type and effect systems – behaviours for concurrency*. Imperial College Press, 1999.

2 Nick Benton, Andrew Kennedy, Martin Hofmann, and Vivek Nigam. Counting successes: Effects and transformations for non-deterministic programs. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 56–72. Springer International Publishing, 2016. doi:10.1007/978-3-319-30936-1_3.

3 Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. doi:10.1093/logcom/2.4.511.

4 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a Formal Theory of Graded Monads. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016*, volume 9634 of *Lecture Notes in Computer Science*, pages 513–530. Springer, 2016. doi:10.1007/978-3-662-49630-5_30.

5 David K. Gifford and John M. Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, August 4-6, 1986, Cambridge, Massachusetts, USA.*, pages 28–38. ACM, 1986. URL: https://dl.acm.org/citation.cfm?id=319838.

6 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2017.13.

7 Pierre Jouvelot and David K Gifford. *Communication effects for message-based concurrency*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

8 John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7:305–317, 1977. doi:10.1007/BF00290339.

9 Shin-ya Katsumata. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 633–645. ACM, 2014. doi:10.1145/2535838.2535846.

10 Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis – Theory and Practice*. CRC Press, 2009. URL: http://www.crcpress.com/product/isbn/9780849328800.

11 Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292 – 310, 2006. Applied Semantics. URL: http://www.sciencedirect.com/science/article/pii/S0304397506005524, doi:https://doi.org/10.1016/j.tcs.2006.08.013.

12 John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. doi:10.1145/367177.367199.

13 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

14 Steven Muchnick and Neil Jones. *Program Flow Analysis: Theory and Applications*. New York University, Courant Institute of Mathematical Sciences, Computer Science Department, January 1981.

15 Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. Effect Systems Revisited - Control-Flow Algebra and Semantics. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi – Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2016. doi:10.1007/978-3-319-27810-0_1.

16 Flemming Nielson, Patrick Cousot, Mads Dam, Pierpaolo Degano, Pierre Jouvelot, Alan Mycroft, and Bent Thomsen. Logical and operational methods in the analysis of programs and

systems. In *LOMAPS workshop on Analysis and Verification of Multiple-Agent Languages*, pages 1–21. Springer, 1996. doi:10.1007/3-540-62503-8_1.

17 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

18 Peter W. O'Hearn. A Primer on Separation Logic (and Automatic Program Verification and Analysis). 33:286–318, 2012. doi:10.3233/978-1-61499-028-4-286.

19 Dominic Orchard and Tomas Petricek. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24, New York, NY, USA, 2014. ACM. doi:10.1145/2633357.2633368.

20 Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014. URL: http://arxiv.org/abs/1401.5391, arXiv:1401.5391.

21 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995*, pages 49–61. ACM Press, 1995. doi:10.1145/199448.199462.

22 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven Muchnick and Neil Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. New York University, Courant Institute of Mathematical Sciences, Computer Science Department, January 1981.

23 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. pages 387–398, 2013. doi:10.1145/2491956.2491978.

24 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

25 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 60–73, 2016. doi:10.1145/2951913.2951924.

26 Ross Tate. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429074.

27 Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992. URL: http://dl.acm.org/citation.cfm?id=143165, doi:10.1145/143165.143169.

28 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003. doi:10.1145/601775.601776.

A Fixed points and transfer functions

Computing transfer functions allows us to compute data-flow values at program points in imperative programs, as we now show.

Let $(N, E, \mathcal{L})$ be a CFG. We consider backwards analyses (forwards ones are similar). For all $n \in N$ there are two associated data-flow values: $DF_{In}(n)$ and $DF_{Out}(n)$, which we here abbreviate to d_n^{In} and d_n^{Out}. Writing ϕ_n for the transfer function for node n, then d_n^{In} satisfies $d_n^{In} = \phi_n(d_n^{Out})$. We also have an equation on d_n^{Out}: if n is an exit node (i.e. no successors), then $d_n^{Out} = \mathrm{BI}$, where BI represents the boundary information ($\emptyset$ both in the cases of liveness and VBE). If n has a non-empty set of successors, then: $d_n^{Out} = \bigsqcup_{s \in succ(n)} d_s^{In}$

Thus for every node in N we have two data-flow equations, say k in total (where $k = 2|N|$). Let $\vec{d}$ be the k-length vector of all data-flow values associated with the nodes, where d_i is its i^{th} component. The data-flow equations give rise to a k-length vector $\vec{h}$ of monotonic functions in $DFValues^k \to DFValues$ such that $d_i = h_i(\vec{d})$, for $1 \leq i \leq k$. The solution of the data-flow equations is the least vector $\vec{d}$ that satisfies these equations. Starting from a k-length vector $(\bot, \ldots, \bot)$, we eventually get to the least solution, given by:

$$\vec{d} = \mathit{fix}\ (\lambda \vec{d'}.\ (h_1(\vec{d'}), \ldots, h_k(\vec{d'})))$$

where $\mathit{fix}(f)$ is the least fixed point of a function f.

By contrast, our approach in this paper is to find monotonic (transfer) functions $H_i : DFValues \to DFValues$ (i.e., on single data-flow values) per node that satisfy $d_i = H_i(\mathrm{BI})$. In this case, we have a function $\hat{H} = \lambda d.\ \langle H_1(d), \ldots, H_k(d)\rangle$ that maps to a tuple of data-flow values, given by the least fixed point:

$$\hat{H} = \mathit{fix}(\lambda \hat{H'}.\ \lambda d.\ \langle h_1(H'_1(d), \ldots, H'_k(d)), \ldots, h_k(H'_1(d), \ldots, H'_k(d))\rangle)$$

In this paper, the effects correspond to a subset of the H_i functions (either just the incoming ones or the outgoing ones), which end up being transfer functions of the *continuations* (or 'from this point on in the CFG') – we can refer to them as *cumulative* transfer functions. The least fixed point $\hat{H}$ in this expression satisfies $d_i = H_i(\mathrm{BI})$ by definition of fixed points. As $d_i = h_i(\vec{d})$ as well, the two approaches give the same result by least fixed point uniqueness.

The main application of these facts is in the proof of Theorem 1. By the conventional definition of live variables, the data-flow equations for the CFG language in this paper are:

$$\begin{array}{lcl}
\mathit{Live}(\ell : \mathsf{halt}\ v) & = & \mathit{fv}(v) \\
\mathit{Live}(\ell : \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell_1\ \mathsf{else\ goto}\ \ell_2) & = & \mathit{fv}(v) \cup \mathit{Live}(\ell_1 : \mathcal{L}(\ell_1)) \cup \mathit{Live}(\ell_2 : \mathcal{L}(\ell_2)) \\
\mathit{Live}(\ell : X := e;\ \mathsf{goto}\ \ell') & = & \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}(\mathit{Live}(\ell' : \mathcal{L}(\ell')))
\end{array}$$

where $\mathit{Live}(\ell : S)$ is the set of live variables at label ℓ corresponding to statement S and $\mathit{fv}(v)$ is the set of free variables in v. This definition is recursive; the set $\widehat{\mathit{Live}}$ corresponds to its least solution in the partial order of sets (that is, subsets of the set of all variables in a CFG).

Proof of Theorem 1. Let there be n instructions labelled $\ell_1, \ldots, \ell_n$ in the CFG. We want to show that $\hat{\Phi}(\ell)(\emptyset) = \widehat{\mathit{Live}}(\ell : \mathcal{L}(\ell))$ hold for all ℓ. For any label ℓ, the exact link between $\hat{\Phi}(\ell)$ and all the other $\hat{\Phi}(\ell')$ depends on what exactly the instruction at ℓ is.

- For $\mathcal{L}(\ell) = \mathsf{halt}\ v$, we have $\hat{\Phi}(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} = \lambda d.\ d \cup \mathit{fv}(v)$.
- For $\mathcal{L}(\ell) = \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else\ goto}\ \ell''$, we have $\hat{\Phi}(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\hat{\Phi}(\ell') \sqcup \hat{\Phi}(\ell''))$, that is, $\hat{\Phi}(\ell) = \lambda d.\ \mathit{fv}(d) \cup \hat{\Phi}(\ell')(d) \cup \hat{\Phi}(\ell'')(d)$.
- For $\mathcal{L}(\ell) = X := e;\ \mathsf{goto}\ \ell'$, we get $\hat{\Phi}(\ell) = \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} \rhd \hat{\Phi}(\ell')$, that is, we have $\hat{\Phi}(\ell) = \lambda d.\ \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}(\hat{\Phi}(\ell')(d))$.

Looking at the previous discussion, $\mathcal{L}(\ell)$ corresponds exactly to $\hat{H}$ as it is also a least solution. Similarly, the expressions for $\widehat{Live}$ are analogous to $\vec{d}$ when looking at the vector given by $\widehat{Live}(\ell_1 : \mathcal{L}(\ell_1)), \ldots, \widehat{Live}(\ell_n : \mathcal{L}(\ell_n))$. The empty set is the bottom element of the data-flow lattice for liveness, so $\mathrm{BI} = \emptyset$. Thus $\widehat{Live}(\ell_i : \mathcal{L}(\ell_i))(\emptyset) = \hat{\Phi}(\ell_i)$ for all $1 \leq i \leq n$. ◀

B Haskell embedding

Modern Haskell as provided by the Glasgow Haskell Compiler (GHC) (from at least version 8.2 onwards) can embed our graded monads with transfer-function effect algebras in its types, leveraging our graded monadic McCarthy transformation. Graded monads can be captured via the following type class which uses type families to provide the grading pomonoid (based on the `effect-monad` package[6] by Orchard et al. [19]):

```
import Prelude hiding (Monad(..)) -- hide regular monads and then...
import qualified Prelude as M     -- ...import as qualified to wrap monads later

class GradedMonad (m :: d → * → *) where      -- Pomonoid graded monads
   type Unit m :: d                           -- {Type-level monoid providing the
   type Seq m (r :: d) (s :: d) :: d          -- effect algebra over domain ‘d‘}
   type Sub m (r :: d) (s :: d) :: Constraint -- Type-level partial order
   -- Graded monad operations
   return :: a → m (Unit m) a
   (>>=) :: m r a → (a → m s b) → m (Seq m r s) b
   sub :: Sub m r s ⇒ m r a → m s a
```

We show the encoding of the compositional live-variable analysis, which is graded by the effect algebra of transfer functions. It is a commonly held belief that type-level functions in Haskell cannot be partially applied, mainly because a type-family based encoding is considered. We show an alternate approach that is much more flexible and suits out purposes well.

To capture transfer functions at the type level, we use a class-based encoding with `d` = `[Symbol]` → `[Symbol]` → `Constraint` meaning that transfer functions are functional relations between two type-level lists of symbols (which are used to represent sets of variables). These type-level lists later get normalised to form sets by removing duplicates and giving an arbitrary consistent ordering, leveraging the `type-level-sets` package.[7]

The gen_v and $kill_v$ functions for variable `v` are defined at the type-level as:

```
class Gen (v :: Symbol) (dIn :: [Symbol]) (dOut :: [Symbol]) | v dIn → dOut
instance Gen v dIn (v ': dIn)                 -- add ‘v‘ to the incoming set ‘dIn‘
class Kill (v :: Symbol) (dIn :: [Symbol]) (dOut :: [Symbol]) | v dIn → dOut
instance Remove dIn v dOut ⇒ Kill v dIn dOut -- rem ‘v‘ from ‘dIn‘ to get ‘dOut‘
```

Classes are types of kind `Constraint` so `Gen v` :: `[Symbol]` → `[Symbol]` → `Constraint`. The syntax `v dIn` → `dOut` is a *functional dependency* telling the type checker that `v` and `dIn` uniquely determine `dOut`, i.e., these class-based relations are really functions. The single instances of each class are then equivalent to the usual λ-based definitions of gen_v and $kill_v$.

The definition of `Kill` uses a recursive type-level function for removing an element from a list, again encoded as a functional relation (for brevity, we skip its recursive definition):

```
class Remove (xs :: [Symbol]) (x :: Symbol) (ys :: [Symbol]) | xs x → ys
```

[6] https://hackage.haskell.org/package/effect-monad
[7] https://hackage.haskell.org/package/type-level-sets

We can capture type-level identity and function composition (which we write as `:|>` due to its later use for the effect algebra) as:

```
class Id dIn dOut | dIn → dOut      -- Identity function
instance Id d d
class (:|>) f g dIn dOut            -- Function composition
instance (f dIn dMid, g dMid dOut) ⇒ (:|>) g f dIn dOut
```

We then define a data type for a Haskell implementation of the graded monad $\mathsf{MultiState}_{\mathsf{triv}}$ by wrapping a monad transformer stack of state monad transformers capturing enough variables for our program. Here we capture a maximum of four mutable variables as:

```
data MultiState (r :: [Symbol] → [Symbol] → Constraint) (a :: Type) =
  MultiState {unMS :: StateT Int (StateT Int (StateT Int (StateT Int Identity))) x}
```

We give `MultiState` a graded monad instance which uses the above type-level identity and function composition:

```
instance GradedMonad MultiState where
   type Unit MultiState     = Id
   type Seq  MultiState r s = r :|> s
   type Sub  MultiState r s = PointwiseSub r s

   return x = MultiState $ M.return x
   (MultiState x) >>= k = MultiState ((M.>>=) x (unMS ∘ k))
   sub (MultiState x) = MultiState x
```

The operations wrap the underlying monad, packing and unpacking the wrapper data type via its constructor and deconstructor. We then define `get` and `put` operations for each of the variables we need, e.g. for X we have `"x"` as its type-level symbol representation:

```
getX :: MultiState (Gen "x") Int
getX = MultiState get
```

```
putX :: Int → MultiState (Kill "x") ()
putX x = MultiState (put x)
```

Example 3 showed the translation of $z := x + y$ as a prefix for a program labelled g. In our Haskell implementation, we can write exactly the same code:

```
exm3 g = do { x ← getX; y ← getY; putZ (x + y); g }
```

This leverages GHC's `RebindableSyntax` extension which allows do {} to be desugared into graded monad operations instead of monad operations. We can then query GHC's type inference which yields the type:

```
exm3 :: MultiState s b → MultiState (Gen "x" :|> (Gen "y" :|> (Kill "z" :|> s))) b
```

To get the data-flow at the current program point, we apply the transfer function grade to the empty set (Section 4.3.1) via the following function:

```
atProgramPoint :: r '[] dOut ⇒ MultiState r x → Set (AsSet dOut)
atProgramPoint (MultiState _) = Set
```

where `AsSet` normalises the type-level list into a set representation and `Set` is a data type with a *phantom type* parameter (not used in any data constructor).

Thus `atProgramPoint` captures the resulting data-flow value `dOut` as a type-level set by forcing the data-flow value input to unify with the boundary value (empty set `'[]`). Applied to `exm3`, GHC calculates the following type representing the set $\{x, y\}$ as expected:

```
atProgramPoint (exm3 (return ())) :: Set '["x", "y"]
```

C Details and proofs for the graded monad of liveness

The state-management operations used in the graded monad definition of Section 4.3.2 (which were omitted for brevity) are defined in turn as follows:

$$\downarrow_{\phi,\phi'} : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \to \mathsf{Store}(\mathit{reads}(\phi)) = \lambda s.s_{|\phi(\emptyset)}$$

i.e., we restrict the domain of the incoming store s to the set $\mathit{reads}(\phi)$ (hence the name of *restriction* for this operator). This relies on the property that $x \in \phi(\emptyset) \implies x \in (\phi \rhd \phi')(\emptyset)$ which is proved by induction on the generating set of transfer functions (see supplement).

$$\blacktriangleleft_{\phi,\phi'} : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \times \mathsf{Store}(\mathit{footprint}(\phi)) \to \mathsf{Store}(\mathit{reads}(\phi'))$$
$$= \lambda(s, s').\{\begin{cases} x \mapsto s'(x) & x \in \mathit{footprint}(\phi) \\ x \mapsto s(x) & x \in \mathit{reads}(\phi \rhd \phi') \wedge x \notin \mathit{footprint}(\phi) \end{cases}\}$$

where $x \in \mathit{reads}(\phi')$ i.e., choose from the right state s' if x is in its domain, otherwise chose from s if x is in its domain but not in the domain of s'. We have that $x \notin \mathit{footprint}(\phi) \wedge x \notin \mathit{reads}(\phi \rhd \phi') \implies x \notin \mathit{reads}(\phi')$ (by induction on generating set of transfer functions) which implies that the resulting map is well-defined (a total function).

$$\triangleleft_{\phi,\phi'} : \mathsf{Store}(\mathit{footprint}(\phi)) \times \mathsf{Store}(\mathit{footprint}(\phi')) \to \mathsf{Store}(\mathit{footprint}(\phi \rhd \phi'))$$
$$= \lambda(s, s').\{\begin{cases} x \mapsto s'(x) & x \in \mathit{footprint}(\phi') \\ x \mapsto s(x) & x \in \mathit{footprint}(\phi) \wedge x \notin \mathit{footprint}(\phi') \end{cases}\}$$

where $x \in \mathit{footprint}(\phi \rhd \phi')$. This merging operator resembles $\blacktriangleleft$, where an additional lemma $x \notin \mathit{footprint}(\phi) \wedge x \notin \mathit{footprint}(\phi') \implies x \notin \mathit{footprint}(\phi \rhd \phi')$ (by induction on generating set of transfer functions) implies that the resulting map is well-defined (a total function).

▶ **Proposition 5** (Restriction right unit). *$\forall \phi$ and $s \in \mathsf{Store}(\mathit{reads}(\phi))$ then $\downarrow_{\phi,id} s \equiv s$*

▶ **Proposition 6** (Merge $\triangleleft$ right unit). *$\forall \phi$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi))$ then $s \triangleleft_{\phi,id} \hat{\emptyset} \equiv s$*

▶ **Proposition 7** (Merge $\blacktriangleleft$ right unit). *$\forall \phi'$ and $s \in \mathsf{Store}(\mathit{reads}(\phi'))$ then $s \blacktriangleleft_{id,\phi'} \hat{\emptyset} \equiv s$*

▶ **Proposition 8** (Merge $\triangleleft$ left unit). *$\forall \phi'$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi'))$ then $\hat{\emptyset} \triangleleft_{id,\phi'} s \equiv s$*

▶ **Proposition 9** (Restriction closure). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ then: $\downarrow_{\phi,\phi'} (\downarrow_{\phi \rhd \phi',\phi''} s) \equiv \downarrow_{\phi,\phi' \rhd \phi''} s$*

▶ **Proposition 10** (Merge $\triangleleft$ associativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi))$, $s' \in \mathsf{Store}(\mathit{footprint}(\phi'))$, and $s'' \in \mathsf{Store}(\mathit{footprint}(\phi''))$ then: $(s \triangleleft_{\phi,\phi'} s') \triangleleft_{(\phi \rhd \phi'),\phi''} s'' \equiv s \triangleleft_{\phi,\phi' \rhd \phi''} (s' \triangleleft_{\phi',\phi''} s'')$.*

▶ **Proposition 11** (Merge $\triangleleft$/$\blacktriangleleft$ associativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ and $s' \in \mathsf{Store}(\mathit{footprint}(\phi))$ and $s'' \in \mathsf{Store}(\mathit{footprint}(\phi'))$ then: $s \blacktriangleleft_{(\phi \rhd \phi'),\phi''} (s' \triangleleft_{\phi,\phi'} s'') \equiv (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s') \blacktriangleleft_{\phi',\phi''} s''$*

▶ **Proposition 12** (Merge $\blacktriangleleft$/restriction commutativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ and $s' \in \mathsf{Store}(\mathit{footprint}(\phi))$ then: $(\downarrow_{\phi \rhd \phi',\phi''} s) \blacktriangleleft_{\phi,\phi'} s' \equiv \downarrow_{\phi',\phi''} (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s')$*

The supplementary material (`https://doi.org/10.5281/zenodo.3784967`) provides the proofs. We now prove the identity and associativity axioms for the graded monad. We refer to the monoid axioms as *idL* ($\mathit{id} \rhd \phi = \phi$) and *idR* ($\phi \rhd \mathit{id} = \phi$) and *assoc* ($(\phi \rhd \phi') \rhd \phi'' = \phi \rhd (\phi' \rhd \phi'')$).

(right identity) $\forall m : M^{\phi}\alpha$ then: $m \gg\!=_{\phi,id} \mathsf{return} \equiv m : M^{\phi}\alpha$ which follows by:

$$
\begin{aligned}
& m \gg\!=_{\phi,id} \mathsf{return} \\
\{defs.{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id} s)\ \mathsf{in}\ \mathsf{let}(z,s'') = ((\lambda x.\lambda s.(x,\hat{\emptyset}))\ y)(s \blacktriangleleft_{\phi,id} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id} s)\ \mathsf{in}\ \mathsf{let}(z,s'') = ((\lambda s.(y,\hat{\emptyset}))(s \blacktriangleleft_{\phi,id} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id} s)\ \mathsf{in}\ \mathsf{let}(z,s'') = (y,\hat{\emptyset})\ \mathsf{in}\ (z, s' \triangleleft_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id} s)\ \mathsf{in}\ (y, s' \triangleleft_{\phi,id} \hat{\emptyset}) \\
\{idR{+}P.5\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m\ s\ \mathsf{in}\ (y, s' \triangleleft_{\phi,id} \hat{\emptyset}) \\
\{idR{+}P.6\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m\ s\ \mathsf{in}\ (y, s') \\
\{\beta{+}\eta\} \equiv\ & m
\end{aligned}
$$

(left identity) $\forall x : \alpha,\ f : \alpha \to M^{\phi'}\beta$ then $\mathsf{return}\ x \gg\!=_{id,\phi'} f = f\ x : M^{\phi'}\beta$ follows by:

$$
\begin{aligned}
& \mathsf{return}\ x \gg\!=_{id,\phi'} f \\
\{defs.{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = ((\lambda x.\lambda s.(x,\hat{\emptyset}))x)(\downarrow_{id,\phi'} s)\ \mathsf{in}\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{id,\phi'} s')\ \mathsf{in}\ (z, s' \triangleleft_{id,\phi'} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = (x,\hat{\emptyset})\ \mathsf{in}\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{id,\phi'} s')\ \mathsf{in}\ (z, s' \triangleleft_{id,\phi'} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)(s \blacktriangleleft_{id,\phi'} \hat{\emptyset})\ \mathsf{in}\ (z, \hat{\emptyset} \triangleleft_{id,\phi'} s'') \\
\{idL{+}P.7\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)s\ \mathsf{in}\ (z, \hat{\emptyset} \triangleleft_{id,\phi'} s'') \\
\{idL{+}P.8\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)s\ \mathsf{in}\ (z, s'') \\
\{\beta{+}\eta\} \equiv\ & f\ x
\end{aligned}
$$

(associativity) $\forall m : M^{\phi}\alpha, f : \alpha \to M^{\phi'}\beta, g : \beta \to M^{\phi''}\gamma$ then: $(m \gg\!=_{\phi,\phi'} f) \gg\!=_{\phi\triangleright\phi',\phi''} g = m \gg\!=_{\phi,\phi'\triangleright\phi''} (\lambda x.\ f\ x \gg\!=_{\phi',\phi''} g)$ follows by:

$$
\begin{aligned}
& (m \gg\!=_{\phi,\phi'} f) \gg\!=_{\phi\triangleright\phi',\phi''} g \\
\{defs{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = \left(\begin{array}{l}(\lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'} s)\ \mathsf{in} \\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{\phi,\phi'} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi,\phi'} s''))\end{array}\right)(\downarrow_{\phi\triangleright\phi',\phi''} s) \\
& \quad \mathsf{in}\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi\triangleright\phi',\phi''} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi\triangleright\phi',\phi''} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = \left(\begin{array}{l}\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'} (\downarrow_{\phi\triangleright\phi',\phi''} s))\ \mathsf{in} \\ \mathsf{let}(z,s'') = (f\ y)((\downarrow_{\phi\triangleright\phi',\phi''} s) \blacktriangleleft_{\phi,\phi'} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi,\phi'} s'')\end{array}\right) \\
& \quad \mathsf{in}\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi\triangleright\phi',\phi''} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi\triangleright\phi',\phi''} s'') \\
\{let{\text{-}}assoc\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'} (\downarrow_{\phi\triangleright\phi',\phi''} s))\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (f\ y)((\downarrow_{\phi\triangleright\phi',\phi''} s) \blacktriangleleft_{\phi,\phi'} s') \\
& \quad \mathsf{let}(z',s''') = (g\ z)(s \blacktriangleleft_{(\phi\triangleright\phi'),\phi''} (s' \triangleleft_{\phi,\phi'} s''))\ \mathsf{in}\ (z', (s' \triangleleft_{\phi,\phi'} s'') \triangleleft_{(\phi\triangleright\phi'),\phi''} s''') \\
\{assoc{+}P.9\text{-}12\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'\triangleright\phi''} s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (f\ y)(\downarrow_{\phi',\phi''} (s \blacktriangleleft_{\phi,\phi'\triangleright\phi''} s'))\ \mathsf{in} \\
& \quad \mathsf{let}(z',s''') = (g\ z)((s \blacktriangleleft_{\phi,\phi'\triangleright\phi''} s') \blacktriangleleft_{\phi',\phi''} s'')\ \mathsf{in} (z', s' \triangleleft_{\phi,\phi'\triangleright\phi''} (s'' \triangleleft_{\phi',\phi''} s''')) \\
\{let{\text{-}}assoc\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'\triangleright\phi''} s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = \left(\begin{array}{l}\mathsf{let}(y,s'') = (f\ y)(\downarrow_{\phi',\phi''} (s \blacktriangleleft_{\phi,\phi'\triangleright\phi''} s'))\ \mathsf{in} \\ \mathsf{let}(z,s''') = (g\ y)((s \blacktriangleleft_{\phi,\phi'\triangleright\phi''} s') \blacktriangleleft_{\phi',\phi''} s'')\ \mathsf{in}\ (z, s'' \triangleleft_{\phi',\phi''} s''')\end{array}\right)\ \mathsf{in} \\
& \quad (z, s' \triangleleft_{\phi,\phi'\triangleright\phi''} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'\triangleright\phi''} s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (\left(\begin{array}{l}\lambda x.\lambda s.\mathsf{let}(y,s') = (f\ x)(\downarrow_{\phi',\phi''} s)\ \mathsf{in} \\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi',\phi''} s')\ \mathsf{in}\ (z, s' \triangleleft_{\phi',\phi''} s'')\end{array}\right)\ y)(s \blacktriangleleft_{\phi,\phi'\triangleright\phi''} s')\ \mathsf{in} \\
& \quad (z, s' \triangleleft_{\phi,\phi'\triangleright\phi''} s'') \\
\{defs{+}\beta\} \equiv\ & m \gg\!=_{\phi,\phi'\triangleright\phi''} (\lambda x.\ f\ x \gg\!=_{\phi',\phi''} g)
\end{aligned}
$$