Predicting Resource Consumption of Higher-Order Workflows

Markus Klinik[1] Jurriaan Hage[2] Jan Martin Jansen[3] Rinus Plasmeijer[1]

[1]Institute for Computing and Information Sciences
Radboud University, Nijmegen, The Netherlands

[2]Department of Information and Computing Sciences
Utrecht University, The Netherlands

[3]Netherlands Defence Academy (NLDA), The Netherlands

m.klinik@cs.ru.nl j.hage@uu.nl jm.jansen.04@mindef.nl rinus@cs.ru.nl

Abstract

We present a type and effect system for the static analysis of programs written in a simplified version of iTasks. iTasks is a workflow specification language embedded in Clean, a general-purpose functional programming language. Given costs for basic tasks, our analysis calculates an upper bound of the total cost of a workflow. The analysis has to deal with the domain-specific features of iTasks, in particular parallel and sequential composition of tasks, as well as the general-purpose features of Clean, in particular let-polymorphism, higher-order functions, recursion and lazy evaluation. Costs are vectors of natural numbers where every element represents some resource, either consumable or reusable.

Categories and Subject Descriptors F.3.3 [*Semantics of Programming Languages*]: Program analysis

Keywords Workflow Systems, Resource Modelling, Type and Effect Systems

1. Introduction

Workflows are algorithmic descriptions of how to combine basic tasks in order to achieve some higher-level goal. The reasons to employ workflows are to understand, define and streamline complicated processes, usually involving many contributing performers. In other words, it is all about optimizing costs.

The most popular workflow systems come in the flavor of Petri-net-like box and arrow diagrams. Most notably there is BPMN (Object Management Group 2009), the Business Process Model and Notation, which is the de-facto industry standard with dozens if not hundreds of software tools from different manufacturers for designing and simulating workflows. From a programming language design perspective BPMN resembles an imperative programming language with GOTOs and barely any capability for abstraction. Control flow is explicit by means of arrows between boxes, but data flow is implicit.

In contrast, iTasks is a workflow management system in the spirit of higher-order functional programming. It comes as a Domain Specific Language embedded in Clean, a lazy functional programming language. As such, workflows written in iTasks can make use of mechanisms common to functional programming, such as higher-order functions, polymorphic static typing, algebraic datatypes and pattern matching. Furthermore, data flow in iTasks is explicit, as tasks can return values. In particular, tasks can take other tasks as input or return them as values, making workflows in iTasks higher-order.

In this paper we present a static analysis for iTasks programs that, given costs for basic tasks, calculates the total cost of a compound workflow. The analysis comes in the form of a type and effect system with polymorphism, polyvariance and subtyping. The analysis is parameterized in the types of resources a workflow uses, which means programmers can define arbitrary units of costs for their workflows.

1.1 Motivating Example

Let us look at an example to get an impression what the analysis can do. Consider the following program.

$$\begin{array}{l}
\textbf{let } \mathit{forever} = \textbf{fix}\, f x.x \gg f x \textbf{ in} \\
\textbf{let } \mathit{approve} = \textbf{fn}\, t.\textbf{use}\ [1\,\text{Supervisor}]\ \textbf{True in} \\
\textbf{let } \mathit{approved} = \textbf{fn}\, t.\mathit{approve}\ t \ggg\!= \textbf{fn}\, \mathit{ok}. \\
\quad \textbf{if } \mathit{ok} \textbf{ then } t \textbf{ else } (\textbf{return } 0) \textbf{ in} \\
\textbf{let } \mathit{initialize} = \textbf{use}\ [3\,\text{Toolboxes}]\ 0 \textbf{ in} \\
\textbf{let } \mathit{work} = \textbf{use}\ [1\,\text{lFuel} + 1\,\text{Toolboxes}]\ 0 \textbf{ in} \\
\mathit{approved}\ \mathit{initialize} \gg \mathit{approved}\ (\mathit{forever}\ \mathit{work})
\end{array}$$

The function *forever* executes its argument task in an endless loop. Approval of a task is represented by *approve*. In a real iTasks program the task t could be presented on screen to a person who then decides whether it should be executed. In this example, tasks are always approved. The function *approved* only executes its argument if it gets approval. The tasks *initialize* and *work* are stand-ins for some tasks where real work happens. They cost 3 toolboxes, and 1 liter of fuel and a toolbox respectively. The main expression first runs initialization and then the actual work in an infinite loop, but only after asking for approval.

The analysis, when run with this program as input, computes as answer the cost $[1\,\text{Supervisor} + \infty\,\text{lFuel} + 3\,\text{Toolboxes}]$. This is because in each iteration of the loop more fuel is consumed, while the toolboxes can be reused. How exactly the analysis computes this answer is the subject of this paper.

The rest of this paper is organized as follows. In section 2 we define syntax and operational semantics of a language to specify workflows. In section 3 we develop the annotated type system that performs the cost analysis. Section 4 describes an algorithm that

PEPM'17, January 16–17, 2017, Paris, France
© 2017 ACM. 978-1-4503-4721-1/17/01...$15.00
http://dx.doi.org/10.1145/3018882.3018885

computes cost estimations. Section 5 discusses the capabilities of the method by means of examples.

2. Syntax and Semantics

In this section we define the syntax and operational semantics of a programming language to specify workflows.

We consider two sorts of resources, *consumables* and *reusables*. Consumable resources are used up when a task that requires them is executed. Reusable resources become available again upon completion, and are claimed exclusively during execution of a task. We assume that it is implicitly understood which resources are consumable and which are reusable.

2.1 A Programming Language for Workflows

Our language is a simplified version of Clean and iTasks. It is a small functional programming language with higher-order functions, non-recursive let-bindings and a fixpoint combinator. Tasks and workflows exist as domain-specific constants and combinators in the language.

$$\begin{gathered} e ::= b \mid i \mid x \mid \mathbf{fn}\, x.e \mid \mathbf{fix}\, f x.e \mid e_1 e_2 \mid \\ \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1 \odot e_2 \mid \\ \mathbf{use}\,[k]\ e \mid \mathbf{return}\ e \mid \\ e_1 \mathbin{\&} e_2 \mid e_1 \ggg\!= e_2 \mid e_1 \gg e_2 \end{gathered}$$

$$k ::= nu \mid nu + k$$

The general-purpose part of the language has Boolean and integer constants b and i, program variables x, abstraction, application, if-then-else and let-bindings. The symbol $\odot$ stands for the usual binary operators for arithmetic, Boolean connectives and comparison. There is a fixpoint combinator $\mathbf{fix}\, f x.e$ that defines recursive functions with one argument.

The domain-specific part of the language has a primitive **use** for basic tasks, and task combinators for sequential and parallel composition. All basic tasks are represented by the **use** operator, where k denotes the cost of executing the task. Costs are given in a polynomial-like syntax where n is a natural number and u the unit of a resource. For example **use** $[2S + 3B]$ 5 may denote a task that when executed uses 2 screwdrivers, 3 bottles of wine, and yields the value 5. Costs and resources are discussed in more detail in Section 2.2.

Expressions of the form **return** e denote tasks that have been executed and return a value e.

Tasks in our language always yield values, but not all real-world tasks do. In monadic programming one usually uses the unit type when values do not matter. For simplicity we do not consider the unit type in our language and just use some integer value that should be ignored by the rest of the program.

There are three combinators for tasks, the parallel combinator (&) and two variants of sequential composition. The regular *bind* operator ($\ggg\!=$) executes its left argument first and passes the resulting value to its right argument as usual. The *sequence* operator ($\gg$) ignores the value of its left argument and yields the value of its right argument. The sequence operator is useful for example programs where the values of tasks do not matter. We include it in our language for convenience, being well aware that it can easily be defined in terms of bind. Since $\gg$ is a variant of $\ggg\!=$ with identical cost behavior, we ignore it in the formal part of this paper but still use it in examples.

The parallel combinator executes both its arguments simultaneously. In iTasks the result value of parallel composition is a tuple containing the values of both tasks. Our language does not have tuples, a deliberate decision because we want to focus more on side effects than on values. Adding tuples would inflate the various definitions relating to our language while not providing substantial new insight regarding cost analysis. We therefore take the liberty of bending the semantics of the parallel combinator a bit to avoid tuples. Our parallel composition still executes both tasks simultaneously, but it discards the value of the left argument and yields the value of the right one.

2.2 A Domain for Representing Costs

In order to define the semantics of the language, we need to make the notions of resource and cost precise.

We define a *resource* to be a positive integral quantity, possibly infinite. Every resource has a unit, so we are not just talking about numbers but about quantities of certain things. Think of liters of fuel or numbers of first-aid kits.

Definition 2.1. The extended natural numbers are denoted by $\overline{\mathbb{N}}$ or $\{0, 1, 2, \ldots, \infty\}$. The only operations we use are addition and comparison, which are extended with ∞ in the obvious way. □

Definition 2.2. Let U be a finite set. A *cost over* U is a function $\gamma : U \to \overline{\mathbb{N}}$. Usually, U is understood from the context and we just talk about *costs*. □

The set U contains the units of the resources of interest in a given workflow. A cost can be seen as a U-indexed set of numbers. For example, let $U = \{mW, l\ \text{water}/h,\ \text{Potatoes}\}$. In a situation like that we are talking about tasks that require any number of these resources. We can then express that a task uses 5 liters of water per hour, 2 potatoes, and no power by associating the following cost to it $[mW \mapsto 0,\ l\ \text{water}/h \mapsto 5,\ \text{Potatoes} \mapsto 2]$. We shall adopt a polynomial-like notation $0mW + 5l/h + 2P$ for this.

Examples in this paper do not mention concrete resources. We are interested in the abstract idea of consumables and reusables, and will use C and R as stand-ins for some consumable and reusable resource respectively. Our example tasks will have costs like $5C + 3R$.

The analysis is based on generating and solving constraints. Constraint solving relies on Tarski's fixpoint theorem, which states that every monotone function on a complete lattice has a least fixpoint. In anticipation of that, the domain of costs must be a complete lattice. We first define the lattice of a single resource.

Lemma 2.3. *The structure* $(\overline{\mathbb{N}}, \leq, \sqcup, \sqcap)$ *is a complete lattice. Joins are maxima, meets are minima. The top element is* ∞*, the bottom element is* 0. □

As said in definition 2.2, a cost is a collection of such numerical quantities. The order on costs comes from the order of the constituents.

Definition 2.4. The set of all possible costs is defined as the function space from U to $\overline{\mathbb{N}}$, extended with a bottom element, written as $[U \to \overline{\mathbb{N}}]_\perp$. □

Definition 2.5. Costs are ordered pointwise. That is, for two costs $\delta, \gamma \in [U \to \overline{\mathbb{N}}]_\perp$

$$\delta \sqsubseteq \gamma \iff \begin{cases} \delta = \perp & \text{or} \\ \forall u \in U. \delta(u) \leq \gamma(u) & \text{if } \delta, \gamma \neq \perp \end{cases}$$ □

In this paper we define several binary operators on costs. All of them are non-strict. This means for all $op \in \{\sqcup, +, \boxplus, \nabla\}$ we implicitly have:

$$\gamma\ op\ \delta = \begin{cases} \gamma & \text{if } \delta = \perp \\ \delta & \text{if } \gamma = \perp \end{cases}$$

The definitions of these operators cover the cases where both γ and $\delta \neq \perp$.

Definition 2.6. Joins of costs are defined pointwise. That is, for two costs δ and γ,

$$\delta \sqcup \gamma = \lambda u.\delta(u) \sqcup \gamma(u).$$

Meets of costs are defined analogously. $\square$

In this paper we work with lattices whose relation we denote by $\sqsubseteq$. This symbol should be read as "below", which implicitly includes "or equal". The inverse relation $\sqsupseteq$ should be read as "above". Similarly, when comparing tasks the terms "cheaper" and "more expensive" implicitly include possible equality.

Lemma 2.7. *Costs form a complete lattice* $([U \to \overline{\mathbb{N}}]_\bot, \sqsubseteq, \sqcup, \sqcap)$. *The top element is* $\lambda u.\infty$*, the bottom element is the artificial* $\bot$*. Joins are pointwise maxima, meets are pointwise minima.* $\square$

This structure will serve as the domain for both the operational semantics and the static analysis. This domain is called $\mathbb{D}$. We use the letters γ and δ to denote elements of $\mathbb{D}$.

We define three binary operations on $\mathbb{D}$ which we use to model sequential composition, parallel composition, and recursion.

Definition 2.8. Addition of costs is defined pointwise.

$$\gamma + \delta = \lambda u.\gamma(u) + \delta(u) \qquad \square$$

Definition 2.9. The *combine* operator $\boxplus$ for costs is defined pointwise as the maximum of reusables and sum of consumables.

$$\gamma \boxplus \delta = \lambda u. \begin{cases} \gamma(u) \sqcup \delta(u) & \text{if } u \text{ is reusable} \\ \gamma(u) + \delta(u) & \text{if } u \text{ is consumable} \end{cases} \qquad \square$$

The combine operator implements the idea that consumables are used up but reusables can be reused. If the left argument already requires 5 units of a reusable resource R, but the right argument only requires 3, then the total requirement is $5R$.

Definition 2.10. Let γ, δ be costs. The *widening of* γ *by* δ, written $\gamma \nabla \delta$, is defined as follows.

$$\gamma \nabla \delta = \lambda u. \begin{cases} \infty & \text{if } \delta(u) > \gamma(u) \\ \gamma(u) & \text{otherwise} \end{cases} \qquad \square$$

We use widening to guarantee termination of fixpoint iteration when the analyzed program contains recursion. The asymmetric definition of widening is intentional. The left argument will always be the overall cost of a recursion and the right argument will always be the cost of a single pass. When the cost of a single pass exceeds the cost of the recursion overall, which means that each iteration adds to the cost, the widening operator yields infinity. Widening is used in the implementation in Section 4 and discussed in more detail in Section 5.

2.3 An Operational Semantics

We split the operational semantics in two parts. Both are call-by-name small-step structural operational semantics with substitution. The general purpose part, denoted by a normal arrow $\to$, applies to non-task expressions. The domain specific part of the semantics applies to task expressions. It is denoted by a two-headed arrow $\twoheadrightarrow$ to emphasize its relation with the bind operator $\ggg=$.

The general purpose semantics rewrites expressions with rules of the form $e \to e$. The domain specific semantics $\langle \gamma, e\rangle \to \langle \gamma, e\rangle$ rewrites expressions while recording the resources that are used during reduction. The names of the rules of the semantics are prefixed by gs- and ds-, which are to be read as "general purpose step" and "domain specific step".

In order to define a small-step semantics for the parallel composition of tasks, we add a new syntactic form to the language, called *cost closure*. When the parallel composition of two tasks needs to be reduced, a cost closure springs into existence and keeps track of the costs of the tasks separately, so that no sharing of resources takes place. Cost closures only exist temporarily during reduction and disappear once both tasks have been executed. Cost closures have the form $par(e_1 \,\&\, e_2, \gamma_1, \gamma_2)$ where $e_1 \,\&\, e_2$ are two tasks in progress of being executed, and γ_1 and γ_2 are their costs so-far.

The general purpose semantics is given in Figure 1. The rule **[gs-fix]** states that a fixpoint reduces to a function where every occurrence of f in the function body is replaced by the fixpoint itself. The rule **[gs-app-cong]** states that in an application, the expression in function position evaluates to a value first. The rule **[gs-app-reduce]** implements call-by-name reduction. The argument is not reduced but substituted as-is. The rule **[gs-let]** implements let-bindings in a call-by-name style. The duplication of costs which results from substituting a task into an expression at several places is intended, because executing a task multiple times costs more. The rule **[gs-if-cong]** states that in a conditional, the condition is evaluated first. The rules **[gs-if-t]** and **[gs-if-f]** reduce to the respective branch if the condition is $\mathbf{True}$ or $\mathbf{False}$.

$$\text{[gs-fix]} \quad \mathbf{fix}\, f x.e \to \mathbf{fn}\, x.e[f \mapsto \mathbf{fix}\, f x.e]$$

$$\text{[gs-app-cong]} \; \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}$$

$$\text{[gs-app-reduce]} \quad (\mathbf{fn}\, x.e_b)e_x \to e_b[x \mapsto e_x]$$

$$\text{[gs-let]} \quad \mathbf{let}\; x = e_x \;\mathbf{in}\; e_b \to e_b[x \mapsto e_x]$$

$$\text{[gs-if-cong]} \; \frac{e_c \to e_c'}{\mathbf{if}\; e_c \;\mathbf{then}\; e_t \;\mathbf{else}\; e_e \to \mathbf{if}\; e_c' \;\mathbf{then}\; e_t \;\mathbf{else}\; e_e}$$

$$\text{[gs-if-t]} \quad \mathbf{if}\;\mathbf{True}\;\mathbf{then}\; e_t \;\mathbf{else}\; e_e \to e_t$$

$$\text{[gs-if-f]} \quad \mathbf{if}\;\mathbf{False}\;\mathbf{then}\; e_t \;\mathbf{else}\; e_e \to e_e$$

Figure 1. Small-step semantics for general purpose expressions

The domain specific semantics is given in Figure 2.

The rule **[ds-use]** stands for the execution of a basic task in a call-by-name style. On the left hand side, γ is the cost of the program so far. Executing the task $\mathbf{use}\;[k]\; e$ uses up the resources k, which is modelled by adding k to γ on the right hand side. The result is $\mathtt{return}\; e$, which indicates that the task has been executed and yields value e.

The rule **[ds-pure]** lifts the general purpose semantics into the domain specific semantics. It is needed for expressions of type *task* that must make a normal step. For example,

$$\langle \mathbf{if}\;\mathbf{True}\;\mathbf{then}\;(\mathbf{use}\;[k]\;5)\;\mathbf{else}\;(\mathbf{use}\;[l]\;7), \gamma\rangle \twoheadrightarrow$$
$$\langle \mathbf{use}\;[k]\;5, \gamma\rangle$$

The rule **[ds-bind-ret]** implements the bind operator. If the left argument of the bind is a task that has been executed, then its result value is passed as an argument to the left argument of the bind.

The rules **[ds-bind-cong]** and **[ds-bind-pure]** state that the left argument of a bind must be reduced first. The cost of the step of e_1 is added to the overall cost of the program.

The rest of the rules all deal with parallel task composition. The only interesting rules are **[ds-para-init]** and **[ds-para-ret]**. [ds-para-init] creates a cost closure in which the parallel composition is executed. When both tasks are fully reduced, which means they are of the form $\mathtt{return}\; e$, [ds-para-ret] calculates the resulting cost such that no resources are shared between the parallel tasks. The result value is the value of the right parallel task, a simplification that, as discussed earlier, saves introduction of tuples to our type system at the cost of deviating from how iTasks works.

The rules **[ds-para-cong-l]**, **[ds-para-cong-r]**, **[ds-para-pure-l]**, and **[ds-para-pure-r]** state how subexpressions must be evaluated so that eventually [ds-para-ret] applies.

$$\text{[ds-use]} \quad \langle \mathbf{use}\,[k]\,e, \gamma\rangle \twoheadrightarrow \langle \mathbf{return}\,e, \gamma \boxplus k\rangle$$

$$\text{[ds-pure]}\ \frac{e \to e'}{\langle e, \gamma\rangle \twoheadrightarrow \langle e', \gamma\rangle}$$

$$\text{[ds-bind-ret]} \quad \langle \mathbf{return}\,e_1 \gg\!= e_2, \gamma\rangle \twoheadrightarrow \langle e_2 e_1, \gamma\rangle$$

$$\text{[ds-bind-cong]}\ \frac{\langle e_1, \bot\rangle \twoheadrightarrow \langle e_1', \gamma'\rangle}{\langle e_1 \gg\!= e_2, \gamma\rangle \twoheadrightarrow \langle e_1' \gg\!= e_2, \gamma \boxplus \gamma'\rangle}$$

$$\text{[ds-bind-pure]}\ \frac{e_1 \to e_1'}{\langle e_1 \gg\!= e_2, \gamma\rangle \twoheadrightarrow \langle e_1' \gg\!= e_2, \gamma\rangle}$$

$$\text{[ds-para-init]} \quad \langle e_1 \,\&\, e_2, \gamma\rangle \twoheadrightarrow \langle \mathit{par}(e_1 \,\&\, e_2, \bot, \bot), \gamma\rangle$$

$$\text{[ds-para-ret]} \quad \begin{array}{l}\langle \mathit{par}(\mathbf{return}\,e_1 \,\&\, \mathbf{return}\,e_2, \gamma_1, \gamma_2), \gamma\rangle \twoheadrightarrow \\ \quad \langle \mathbf{return}\,e_2, \gamma \boxplus (\gamma_1 + \gamma_2)\rangle\end{array}$$

$$\text{[ds-para-cong-l]}\ \frac{\langle e_1, \gamma_1\rangle \twoheadrightarrow \langle e_1', \gamma_1'\rangle}{\begin{array}{l}\langle \mathit{par}(e_1 \,\&\, e_2, \gamma_1, \gamma_2), \gamma\rangle \twoheadrightarrow \\ \langle \mathit{par}(e_1' \,\&\, e_2, \gamma_1', \gamma_2), \gamma\rangle\end{array}}$$

$$\text{[ds-para-cong-r]}\ \frac{\langle e_2, \gamma_2\rangle \twoheadrightarrow \langle e_2', \gamma_2'\rangle}{\begin{array}{l}\langle \mathit{par}(e_1 \,\&\, e_2, \gamma_1, \gamma_2), \gamma\rangle \twoheadrightarrow \\ \langle \mathit{par}(e_1 \,\&\, e_2', \gamma_1, \gamma_2'), \gamma\rangle\end{array}}$$

$$\text{[ds-para-pure-l]}\ \frac{e_1 \to e_1'}{\begin{array}{l}\langle \mathit{par}(e_1 \,\&\, e_2, \gamma_1, \gamma_2), \gamma\rangle \twoheadrightarrow \\ \langle \mathit{par}(e_1' \,\&\, e_2, \gamma_1, \gamma_2), \gamma\rangle\end{array}}$$

$$\text{[ds-para-pure-r]}\ \frac{e_2 \to e_2'}{\begin{array}{l}\langle \mathit{par}(e_1 \,\&\, e_2, \gamma_1, \gamma_2), \gamma\rangle \twoheadrightarrow \\ \langle \mathit{par}(e_1 \,\&\, e_2', \gamma_1, \gamma_2), \gamma\rangle\end{array}}$$

Figure 2. Small-step semantics for task expressions

3. The Analysis

In this section we present a static analysis that estimates the cost of executing a workflow. The estimation is conservative, which means that no execution of a workflow uses more resources than the analysis predicts.

The analysis is realized as a type and effect system with polymorphism, polyvariance and subtyping. The type system collects constraints such that a solution of the constrains gives an estimation of the cost of executing the program.

3.1 The Annotated Type System

The type system has type variables α, two base types for Booleans and integers, and two type constructors for functions and tasks. The only annotated types are tasks, where annotations come in the form of annotation variables β. Annotation variables will be given meaning by the solution of constraint sets. Types are formed by the following grammar.

$$\hat\tau ::= \alpha \mid \mathit{bool} \mid \mathit{int} \mid \mathit{task}\,\beta\,\hat\tau \mid \hat\tau_1 \to \hat\tau_2$$

The type system uses type schemes for polymorphism and polyvariance. Additionally, type schemes are qualified, which means they include constraints relating bound variables. Type schemes are formed by the following grammar.

$$\sigma ::= \hat\tau \mid \forall \vec\alpha \vec\beta. C \Rightarrow \hat\tau$$

In a type scheme, $\vec\alpha$ are bound type variables, $\vec\beta$ bound annotation variables, and C is a set of constraints that restricts bound annotation and type variables.

When $\vec\alpha$ and $\vec\beta$ are empty, which means there are no bound variables in $\hat\tau$, C will usually be empty as well. In this case we just write $\hat\tau$ instead of $\forall().\emptyset \Rightarrow \hat\tau$.

Definition 3.1. (Free type and annotation variables) We define two functions on types and type schemes, FTV and FAV. FTV returns all free type variables of a type scheme, and FAV all free annotation variables. They are defined by induction on the syntax of types. We omit the formal definition. For example

$$\mathit{FAV}(\forall\alpha_1\beta_1.\{\,\beta_2 \sqsupseteq \beta_1\,\} \Rightarrow \mathit{task}\,\beta_1\,\alpha_1 \to \mathit{task}\,\beta_2\,\alpha_2) = \{\,\beta_2\,\}.$$

These functions extend pointwise to environments. We also define them for annotations and constraint sets, where they just return all occurring variables, because there are no bound variables in constraints. □

Judgements in the type system are of the form $C;\Gamma \vdash e : \hat\tau$ where Γ assigns types to free program variables in e. C is a set of constraints that relates free annotation and type variables in $\hat\tau$.

Definition 3.2. (Annotations) Annotations are expressions denoting costs. They are used in constraints to describe the cost behavior of programs. Annotations are formed by the following grammar.

$$\varphi ::= k \mid \beta \mid \varphi_1 + \varphi_2 \mid \varphi_1 \boxplus \varphi_2 \mid \varphi_1 \nabla \varphi_2$$

k stands for cost constants like $[2C + 7R]$. β stands for annotation variables. The three operators $+, \boxplus, \nabla$ stand for their respective operations on $\mathbb{D}$, defined in Section 2.2. □

Definition 3.3. (Constraints) Constraints come in two forms, one that relates types and one that relates costs. Constraints are formed by the following grammar.

$$c ::= \hat\tau_1 <: \hat\tau_2 \mid \beta \sqsupseteq \varphi$$

A constraint of the form $\hat\tau_1 <: \hat\tau_2$ is called a *subsumption constraint* and records the fact that $\hat\tau_1$ must be a subtype of $\hat\tau_2$. A constraint of the form $\beta \sqsupseteq \varphi$, called an *effect constraint*, means that the cost β is above the cost expressed by φ. □

Effect constraints $\beta \sqsupseteq \varphi$ use the symbol "above" and always have a variable on the left, and an annotation on the right hand side. This is customary in literature on type and effect systems, and we adopt it here. Please note that the direction of subsumption constraints is the opposite of effect constraints, again an adoption of convention in the literature.

Definition 3.4. Given an annotation φ, we define the interpretation of φ, written $[\![\varphi]\!]$, as a function from assignments to costs. An assignment $s : \text{AnnVar} \to \mathbb{D}$ is a mapping from annotation variables to costs.

$$\begin{aligned}[\![k]\!]s &= k\\ [\![\beta]\!]s &= s(\beta)\\ [\![\varphi_1 + \varphi_2]\!]s &= [\![\varphi_1]\!]s + [\![\varphi_2]\!]s\\ [\![\varphi_1 \boxplus \varphi_2]\!]s &= [\![\varphi_1]\!]s \boxplus [\![\varphi_2]\!]s\\ [\![\varphi_1 \nabla \varphi_2]\!]s &= [\![\varphi_1]\!]s \nabla [\![\varphi_2]\!]s\end{aligned}$$

Interpretations of annotations are used in the effect constraint solver in Section 4. □

To streamline the discussion about constraints and constraint sets, we introduce the constraint entailment relation, which makes use of the following fact.

Fact 3.5. *Any constraint set C gives rise to a monotone function*

$$F_C : (\mathit{AnnVar}_C \to \mathbb{D}) \to (\mathit{AnnVar}_C \to \mathbb{D}). \qquad \square$$

Monotonicity comes from the fact that the interpretation of annotations, see Definition 3.4, is monotone. AnnVar$_C$ is the set of annotation variables occuring in C. Our domain $\mathbb{D}$ is a complete lattice, and so is the function space AnnVar$_C \to \mathbb{D}$. As such, F_C has a least fixpoint which is at the same time the least solution to C.

Definition 3.6. (Constraint entailment) An assignment s *satisfies* an effect constraint $\beta \sqsupseteq \varphi$ iff $s(\beta) \sqsupseteq [\![\varphi]\!]s$. Let C be a constraint set and c a constraint. We write $C \Vdash c$ if the minimal solution s of C satisfies c. The existence of such a solution is guaranteed by Tarski's fixpoint theorem and Fact 3.5. We write $C \Vdash D$ if every solution of C is a solution of D. Satisfaction of subsumption constraints is defined in Definition 3.7. $\square$

The typing rules for our annotated type system are given in Figure 3. The rule scheme **[t-const]** gives the corresponding types to pure constants, that is *bool* to Boolean constants and *int* to integer constants. **[t-var]** looks up the type scheme of a variable in the environment and instantiates it. **[t-op]** is a rule scheme for all pure operators in the language and typechecks them in the obvious way. Addition takes two integers and returns an integer, comparisons take two integers and return a Boolean, and so on. **[t-fn]** typechecks abstractions in the usual way. The bound variable is put into the environment and the function body is analyzed in this extended environment. **[t-fix]** analyzes recursive functions similarly to [t-fn]. The function body is analyzed under the assumption that f is a function that takes an argument of the type of x. **[t-app]**, the rule for function application, requires that the type of the actual parameter must be a subtype of the type of the formal parameter. The idea is that a function that expects an expensive argument has enough resources to also deal with a cheaper argument. For base types subtyping merely means type equality. **[t-if]** requires the condition to be Boolean, as usual. The overall type of the conditional must be a supertype of the types of both branches. The idea is that if one branch is more expensive than the other, and the context in which the conditional is used has enough resources for the expensive one, it can also deal with the cheaper one. **[t-let]** implements polymorphism and polyvariance. It analyzes the defining expression e_x, generalizes its type, and analyzes the body under the assumption that x has the generalized type. [t-let] plays together with [t-var] to allow every use of x to have a different type, as far as instantiation allows. **[t-return]** states that an expression of the form $\mathbf{return}\ e$ is a task that has no cost. **[t-use]** states that basic tasks have the cost that the programmer gave them. **[t-bind]** states that two tasks, executed in sequence, have the combined cost of the left and the right cost, where reusables can be reused. **[t-para]** states that the parallel composition of two tasks costs as much as the sum of the two tasks individually. No sharing of reusables takes place. **[t-cost-closure]** states how to calculate the cost of two parallel tasks e_1 and e_2, whose execution to the current form already costed γ_1 and γ_2 respectively. β_1 and β_2 are the predicted costs of e_1 and e_2. The overall predicted cost of the cost closure is the sum of the combinations of the actual costs so far and the predicted rest.

Definition 3.7. (Subtyping) Subtyping is the reflexive transitive closure of the relation defined by the rules in Figure 4. Reflexivity in particular means that $\Vdash$ *bool* $<:$ *bool* and $\Vdash$ *int* $<:$ *int*. $\square$

Subtyping expresses that if the types of two expressions $e_1 : \hat{\tau}_1$ and $e_2 : \hat{\tau}_2$ are in the subtype relation $\hat{\tau}_1 <: \hat{\tau}_2$, then e_1 can be used in all the places where e_2 can be used.

$$[\text{t-const}] \quad C;\Gamma \vdash c : \tau_c$$

$$[\text{t-var}] \ \frac{\Gamma(x) \succ D, \hat{\tau} \qquad C \Vdash D}{C;\Gamma \vdash x : \hat{\tau}}$$

$$[\text{t-op}] \ \frac{C;\Gamma \vdash e_1 : \tau_\odot^1 \qquad C;\Gamma \vdash e_2 : \tau_\odot^2}{C;\Gamma \vdash e_1 \odot e_2 : \tau_\odot}$$

$$[\text{t-fn}] \ \frac{C;\Gamma[x \mapsto \hat{\tau}_x] \vdash e_b : \hat{\tau}_b}{C;\Gamma \vdash \mathbf{fn}\, x.e_b : \hat{\tau}_x \to \hat{\tau}_b}$$

$$[\text{t-fix}] \ \frac{C;\Gamma[f \mapsto \hat{\tau}_x \to \hat{\tau}_b][x \mapsto \hat{\tau}_x] \vdash e_b : \hat{\tau}_b}{C;\Gamma \vdash \mathbf{fix}\, f x.e_b : \hat{\tau}_x \to \hat{\tau}_b}$$

$$[\text{t-app}] \ \frac{\begin{array}{ll} C;\Gamma \vdash e_1 : \hat{\tau}_1 \to \hat{\tau}_2 & \\ C;\Gamma \vdash e_2 : \hat{\tau}_3 & C \Vdash \hat{\tau}_3 <: \hat{\tau}_1 \end{array}}{C;\Gamma \vdash e_1 e_2 : \hat{\tau}_2}$$

$$[\text{t-if}] \ \frac{\begin{array}{lll} & C;\Gamma \vdash e_t : \hat{\tau}_t & C \Vdash \hat{\tau}_t <: \hat{\tau} \\ C;\Gamma \vdash e_c : \mathit{bool} & C;\Gamma \vdash e_e : \hat{\tau}_e & C \Vdash \hat{\tau}_e <: \hat{\tau} \end{array}}{C;\Gamma \vdash \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e : \hat{\tau}}$$

$$[\text{t-let}] \ \frac{\begin{array}{l} C';\Gamma \vdash e_x : \hat{\tau}_x \\ C;\Gamma[x \mapsto \mathit{generalize}(\hat{\tau}_x, C', \Gamma)] \vdash e_b : \hat{\tau} \end{array}}{C;\Gamma \vdash \mathbf{let}\ x = e_x\ \mathbf{in}\ e_b : \hat{\tau}}$$

$$[\text{t-return}] \ \frac{C;\Gamma \vdash e : \hat{\tau} \qquad C \Vdash \beta \sqsupseteq \bot}{C;\Gamma \vdash \mathbf{return}\ e : \mathit{task}\, \beta\, \hat{\tau}} \ \beta \text{ fresh}$$

$$[\text{t-use}] \ \frac{C;\Gamma \vdash e : \hat{\tau} \qquad C \Vdash \beta \sqsupseteq k}{C;\Gamma \vdash \mathbf{use}\ [k]\ e : \mathit{task}\, \beta\, \hat{\tau}} \ \beta \text{ fresh}$$

$$[\text{t-bind}] \ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : \mathit{task}\, \beta_1\, \hat{\tau}_1 \\ C;\Gamma \vdash e_2 : \hat{\tau}_1 \to \mathit{task}\, \beta_2\, \hat{\tau}_2 \\ C \Vdash \beta \sqsupseteq \beta_1 \boxplus \beta_2 \end{array}}{C;\Gamma \vdash e_1 \gg\!= e_2 : \mathit{task}\, \beta\, \tau_2} \ \beta \text{ fresh}$$

$$[\text{t-para}] \ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : \mathit{task}\, \beta_1\, \hat{\tau}_1 \\ D;\Gamma \vdash e_2 : \mathit{task}\, \beta_2\, \hat{\tau}_2 \\ C \cup D \Vdash \beta \sqsupseteq \beta_1 + \beta_2 \end{array}}{C \cup D;\Gamma \vdash e_1\ \&\ e_2 : \mathit{task}\, \beta\, \hat{\tau}_2} \ \beta \text{ fresh}$$

$$[\text{t-cost-closure}] \ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : \mathit{task}\, \beta_1\, \hat{\tau}_1 \\ D;\Gamma \vdash e_2 : \mathit{task}\, \beta_2\, \hat{\tau}_2 \\ C \cup D \Vdash \beta \sqsupseteq (\beta_1 \boxplus \gamma_1) + (\beta_2 \boxplus \gamma_2) \\ \beta \text{ fresh} \end{array}}{C \cup D;\Gamma \vdash \mathit{par}(e_1\ \&\ e_2, \gamma_1, \gamma_2) : \mathit{task}\, \beta\, \hat{\tau}_2}$$

Figure 3. The annotated type system

$$\text{[st-task]}\ \frac{C \Vdash \beta_1 \sqsubseteq \beta_2 \qquad C \Vdash \hat{\tau}_1 <: \hat{\tau}_2}{C \Vdash \mathit{task}\, \beta_1\, \hat{\tau}_1 <: \mathit{task}\, \beta_2\, \hat{\tau}_2}$$

$$\text{[st-fun]}\ \frac{C \Vdash \hat{\tau}_a' <: \hat{\tau}_a \qquad C \Vdash \hat{\tau}_r <: \hat{\tau}_r'}{C \Vdash \hat{\tau}_a \to \hat{\tau}_r <: \hat{\tau}_a' \to \hat{\tau}_r'}$$

Figure 4. The subtyping relation

In our system this means that either e_1 and e_2 are of base types, which means $\hat{\tau}_1 = \hat{\tau}_2$, or $\hat{\tau}_1$ is cheaper than $\hat{\tau}_2$. If a context has enough resources for $\hat{\tau}_2$, it also can deal with $\hat{\tau}_1$. This is essentially what the subtyping rule **[st-task]** says.

Example 3.8. Consider the constraint set $C = \{\, \beta_1 \sqsupseteq [1C], \beta_2 \sqsupseteq [2C] \,\}$. The minimal solution s of C has $[\beta_1 \mapsto [1C], \beta_2 \mapsto [2C]]$. By definition of $\Vdash$, this means $C \Vdash \beta_1 \sqsubseteq \beta_2$. This allows us to form the following small derivation, which witnesses the fact that the two task types in the conclusion are in fact in the subtype relation.

$$\text{[st-task]}\ \frac{C \Vdash \beta_1 \sqsubseteq \beta_2 \qquad \Vdash \mathit{int} <: \mathit{int}}{C \Vdash \mathit{task}\, \beta_1\, \mathit{int} <: \mathit{task}\, \beta_2\, \mathit{int}} \qquad \square$$

The rule **[st-fun]** implements subtyping for functions. As usual in systems with subtyping, functions are contravariant in the argument type $\hat{\tau}_a$ and covariant in the result type $\hat{\tau}_r$. Covariant means that the result types of two functions are in the same subtyping order as the functions overall, while contravariant means that the order of the argument types is reversed. To understand the latter, consider two functions f and g with the same result type $\hat{\tau}_r$, but with argument types in relation $\hat{\tau}_f <: \hat{\tau}_g$. In other words f expects cheaper arguments than g. Then, according to [st-fun], $\hat{\tau}_g \to \hat{\tau}_r <: \hat{\tau}_f \to \hat{\tau}_r$, which means g can be used in all contexts in which f can be used and possibly more. The intuitive reason is that if g can deal with expensive arguments, it can also deal with the cheaper arguments of f. The formal reason is that our type system allows any expression to be considered as more expensive if needed. If the application $f\, e$ is well-typed then the type system can make e more expensive so that $g\, e$ is also well typed.

In some rules we require a fresh annotation variable β. Fresh here means that β must not occur in the free annotation variables of the environment: $\beta \notin \mathrm{FAV}(\Gamma)$. The variable β may however occur in other branches of the type derivation.

Definition 3.9. (Generalization) Let Γ be an environment, C a constraint set and $\hat{\tau}$ a type. The generalization of $\hat{\tau}$ with respect to Γ and C is defined as follows.

$$\begin{aligned} &\mathit{generalize}(\hat{\tau}, C, \Gamma) = \forall \vec{\alpha}\vec{\beta}.C \Rightarrow \hat{\tau} \quad \mathit{where} \\ &\quad \vec{\alpha} = FTV(\hat{\tau}) \setminus FTV(\Gamma) \\ &\quad \vec{\beta} = FAV(\hat{\tau}) \setminus FAV(\Gamma) \end{aligned} \qquad \square$$

Definition 3.10. (Instantiation) Let $\sigma = \forall \vec{\alpha}\vec{\beta}.C \Rightarrow \hat{\tau}$ be a type scheme. An *instantiation* of σ is a type τ' and a constraint set C' where type variables are substituted by types, and annotation variables are substituted by fresh annotation variables. As usual, this substitution respects bound variables and is capture-avoiding. We write $\sigma \succ C', \hat{\tau}'$ if $C', \hat{\tau}'$ is an instantiation of σ. □

Example 3.11. Some type schemes and instantiations.

$$\hat{\tau} \succ \emptyset, \hat{\tau}$$
$$\forall \alpha_1 \alpha_2.\{\, \alpha_1 <: \alpha_2 \,\} \Rightarrow \alpha_1 \to \alpha_2 \succ \{\, \alpha_3 <: \alpha_4 \,\}, \alpha_3 \to \alpha_4$$
$$\forall \alpha_1 \alpha_2.\{\, \alpha_1 <: \alpha_2 \,\} \Rightarrow \alpha_1 \to \alpha_2 \succ \{\, \mathit{int} <: \mathit{int} \,\}, \mathit{int} \to \mathit{int} \qquad \square$$

Terminology. Given a program e, an analysis result $C; \Gamma \vdash e : \mathit{task}\, \beta\, \hat{\tau}$, and the least solution s of C, we call $s(\beta)$ the *predicted cost* of e. If $\langle e, \bot \rangle \twoheadrightarrow^* \langle v, \gamma \rangle$, we call γ the *actual cost* of this reduction. We write $s \vDash C$ for the least solution s of C.

Conjecture 3.12. *(Correctness) The analysis is sound with respect to the operational semantics. In other words, the predicted cost of a program is always above the actual cost of any possible reduction. Formally,*

$$\begin{array}{rrcl} \mathit{assume} & C; \Gamma & \vdash & e : \mathit{task}\, \beta\, \hat{\tau} \\ \mathit{and} & \langle e, \bot \rangle & \twoheadrightarrow^* & \langle e', \gamma' \rangle \\ \mathit{and} & C'; \Gamma & \vdash & e' : \mathit{task}\, \beta'\, \hat{\tau}. \\ \mathit{Let} & s & \vDash & C \\ \mathit{and} & s' & \vDash & C'. \\ \mathit{Then} & s(\beta) & \sqsupseteq & s'(\beta') \boxplus \gamma'. \end{array} \qquad \square$$

4. Implementation

This section describes an algorithm that implements the type system. We implemented the algorithm in Clean, but this paper describes it in an abstract way that ignores many implementation details. The source code, together with example programs and unit tests can be found online[1]. The implementation is intended to be a proof-of-concept. No effort has been put into optimization.

The analysis works in three steps. First, there is a modified version of algorithm $\mathcal{W}$ that infers types and collects constraints about type and annotation variables. Second, the subsumption constraint solver decomposes subtype constraints into effect constraints and unifications. Third, and only if the top-level expression is of type *task*, a worklist algorithm calculates a solution of the effect constraints. The cost of the analyzed program is the entry in the solution that corresponds to the annotation variable of the top-level expression. If the top-level expression is not a task, and therefore does not have a definite cost, the analysis just reports the type and the constraints.

Our system deals with two kinds of variables: annotation variables and type variables. Substitutions θ replace type variables by types and annotation variables by annotation *variables*. We take the liberty to combine type and annotation substitutions for the sake of brevity. Type and annotation substitutions are denoted by $[\alpha \mapsto \hat{\tau}]$ and $[\beta_1 \mapsto \beta_2]$ respectively. The empty substitution is denoted by $[\,]$. Substitutions are applicable to types, type schemes and constraints, and extend pointwise to environments and constraint sets. When applied to type schemes, substitutions respect bound variables. Substitutions can be composed, which is denoted by $\theta_2 \circ \theta_1$, and defined as $(\theta_2 \circ \theta_1)\hat{\tau} = \theta_2(\theta_1 \hat{\tau})$.

4.1 Algorithm W

Type inference uses a unification algorithm. Unification takes two types and returns a substitution if the types can be unified, and an error otherwise. Figure 5 shows the unification algorithm $\mathcal{U}$. The difference between unification in textbook Hindley-Milner algorithms and our algorithm is that ours has to deal with task types, which carry annotation variables. As such, unification is also applicable to annotation variables. The relevant clauses are in lines (1) and (2) in Figure 5. In line **(1)**, tasks are unified by unifying their annotation variables and their return types. In line **(2)**, annotation variables are unified by generating a substitution.

The first step of the analysis is a variant of algorithm $\mathcal{W}$. It takes as input a program e in our language and an environment Γ, and returns a triple of an annotated type $\hat{\tau}$, a substitution θ, and a set of constraints C. The returned results are such that if s is a solution of θC, and e is of type *task*, that is $\theta\hat{\tau} = \mathit{task}\, \beta\, \hat{\tau}_1$, then $s(\beta)$ is an upper bound of the cost of e. Algorithm $\mathcal{W}$ is given in Figures 6 and 7.

[1] `https://gitlab.science.ru.nl/mklinik/program-analysis`

$$\begin{aligned}
&\mathcal{U}(\mathit{bool}, \mathit{bool}) = [\,] \\
&\mathcal{U}(\mathit{int}, \mathit{int}) = [\,] \\
&\mathcal{U}(\alpha, \alpha) = [\,] \\
&\mathcal{U}(\alpha, \hat{\tau}) = [\alpha \mapsto \hat{\tau}] \text{ if } \alpha \notin FTV(\hat{\tau}), \text{Error otherwise.} \\
&\mathcal{U}(\hat{\tau}, \alpha) = \mathcal{U}(\alpha, \hat{\tau}) \\
&\mathcal{U}(\hat{\tau}_1 \to \hat{\tau}_2, \hat{\tau}_3 \to \hat{\tau}_4) = \theta_2 \circ \theta_1 \text{ where} \\
&\quad \theta_1 = \mathcal{U}(\hat{\tau}_1, \hat{\tau}_3) \\
&\quad \theta_2 = \mathcal{U}(\theta_1\hat{\tau}_2, \theta_1\hat{\tau}_4) \\
&\mathcal{U}(\mathit{task}\,\beta_1\,\hat{\tau}_1, \mathit{task}\,\beta_2\,\hat{\tau}_2) = \theta_2 \circ \theta_1 \text{ where} && (1) \\
&\quad \theta_1 = \mathcal{U}(\beta_1, \beta_2) \\
&\quad \theta_2 = \mathcal{U}(\theta_1\hat{\tau}_1, \theta_1\hat{\tau}_2) \\
&\mathcal{U}(\hat{\tau}_1, \hat{\tau}_2) = \text{Error, cannot unify types.} \\
&\mathcal{U}(\beta, \beta) = [\,] \\
&\mathcal{U}(\beta_1, \beta_2) = [\beta_1 \mapsto \beta_2] && (2)
\end{aligned}$$

Figure 5. The unification algorithm.

Figure 6 shows type inference for pure expressions. Line **(1)** and **(2)** typecheck Boolean and integer constants. Such expressions are of type *bool* and *int* respectively.

The clause for variables in line **(3)** looks up the type scheme of x in the environment and instantiates it. Instantiation means that fresh type- and annotation variables are generated and substituted for all the bound ones in the type and the constraint set.

The clause for abstractions in line **(4)** puts the bound variable x into the environment with a fresh type variable α and typechecks the function body.

Recursive functions, line **(5)**, are typechecked by putting the function and the argument with fresh type variables into the environment and then checking the function body. The resulting constraints are widened using the function widen.

Definition 4.1. (Widening) The function widen takes a set of constraints and yields a set of constraints where the variable in each constraint is widened with its own right hand side. Subsumption constraints are left untouched. Formally:

$$\begin{aligned}\mathit{widen}(C) &= \{\, \beta \sqsupseteq \beta \mathbin{\nabla} \varphi \mid \beta \sqsupseteq \varphi \in C \,\} \\ &\cup \{\, \hat{\tau}_1 <: \hat{\tau}_2 \mid \hat{\tau}_1 <: \hat{\tau}_2 \in C \,\}\end{aligned} \qquad \square$$

Finally, unification of the type of the body and the return type of the function makes sure that all acquired information is contained in the result. This clause is the only place where widening is applied, because it is the main source of recursive constraints. Unfortunately there are other situations in which recursive constraints can arise, as discussed in Example 5.10.

The clause for applications, line **(6)**, typechecks function and argument expressions independently and uses unification to make sure that the function expression has function type. In contrast to textbook Hindley-Milner, the clause does not unify the types of the formal and actual arguments. Instead it generates a subtyping constraint that requires the actual argument to be a subtype of the formal argument.

The clause for conditionals, line **(7)**, typechecks the condition and uses unification to make sure that it is of type *bool*. It then typechecks the then- end else-branches independently and generates two subtyping constraints which make sure that the type of the conditional is a supertype of both branches.

The clause for let-bindings in line **(8)** first typechecks the defining expression of x and then generalizes the type and constraints resulting from that. The body of the let-binding is then typechecked in an environment where x maps to its type scheme.

The clause for pure operators, line **(9)**, does not deal with annotations at all. It typechecks both arguments and uses unification to make sure that the actual argument types match the formal argument types of the operator.

The rest of the algorithm is shown in Figure 7.

The clause in line **(10)** handles task constants. It generates a constraint that makes sure that the constraint solver takes the cost of the task into account.

The clause for sequential task composition, line **(11)**, looks complex but holds no surprises. The left argument must be a task and the right argument a function that accepts the task's value. The function must yield a task. The clause generates a constraint that ensures that the cost of the overall expression is the combination of the cost of the left and right arguments.

Parallel task composition, line **(12)**, is simpler than sequential composition. Both arguments must be tasks. The value of the overall expression is the value of the right argument. See Section 2.1 for a rationale. The cost of the overall expression is the sum of the costs of the arguments.

4.2 Subsumption Constraint Solving

This section motivates and describes the subsumption constraint solver.

In many type and effect systems, all types are annotated with effects. An example is exception analysis, where expressions of any type can throw exceptions. In our case however, only tasks have annotations because only tasks have costs.

Our algorithm $\mathcal{W}$ performs resource analysis by traversing the abstract syntax tree. This always happens in a particular order, and as with Hindley-Milner type inference we face the problem that information about the type of some expressions becomes available only when the algorithm advances further into the AST.

Hindley-Milner solves this by generating and solving type equality constraints. Our system features subtyping, which means that unification would lead to faulty results. Instead of type equality constraints, we must generate subeffect constraints, but only if the involved types are tasks. For base types we want regular unification. If the algorithm cannot immediately determine whether to use unification or subeffecting, it has to defer the decision until more information about the types is available.

Generating subsumption constraints, which are later resolved by either unification or turning them into subeffect constraints, is conceptually similar to annotating all types and dropping the annotation when it becomes clear that a type is not a task. We chose for the former because it allows the implementation of different forms of subtyping and subeffecting just by modifying the constraint solver. The constraint solver presented in this paper implements full subtyping.

Figure 8 shows the subsumption constraint solver. Subsumption constraint solving happens after algorithm $\mathcal{W}$, and its goal is to decompose all subsumption constraints as far as possible to extract effect constraints according to the subtyping rules.

Subsumption solving is a many-pass procedure over the list of constraints that algorithm $\mathcal{W}$ collects. In every pass unification can take place, in which case the solver learns more about the type variables in the constraint set. As long as the solver learns more details, it needs to continue performing passes. The procedure eventually terminates because the types involved in subsumption constraints become structurally smaller in each pass, and our system does not have recursive types.

Figure 8 shows the function $\mathit{solveSubsumption}$, which handles a single constraint. This function is repeatedly called in one pass, and a top-level loop, described but not shown in this paper, performs as many passes as necessary. The function $\mathit{solveSubsumption}$ gets a single constraint as input and returns a tuple consisting of effect

$$\mathcal{W}(\Gamma, \mathbf{True}) = \mathcal{W}(\Gamma, \mathbf{False}) = \langle \mathit{bool},\ [\,],\ \emptyset \rangle \tag{1}$$

$$\mathcal{W}(\Gamma, n) = \langle \mathit{int},\ [\,],\ \emptyset \rangle \tag{2}$$

$$\mathcal{W}(\Gamma, x) = \langle \hat{\tau},\ [\,],\ C \rangle \text{ where} \tag{3}$$
$$\langle \hat{\tau}, C \rangle = \mathit{inst}(\Gamma(x))$$

$$\mathcal{W}(\Gamma, \mathbf{fn}\, x.e_b) = \langle \theta_b\alpha \to \hat{\tau}_b,\ \theta_b,\ C_b \rangle \text{ where} \tag{4}$$
$$\alpha \text{ fresh}$$
$$\langle \hat{\tau}_b, \theta_b, C_b \rangle = \mathcal{W}(\Gamma[x \mapsto \alpha], e_b)$$

$$\mathcal{W}(\Gamma, \mathbf{fix}\, f x.e_b) = \langle (\theta_2 \circ \theta_1)\alpha_x \to \theta_2\hat{\tau}_1,\ \theta_2 \circ \theta_1,\ \theta_2 C_1' \rangle \tag{5}$$
$$\text{where } \alpha_x, \alpha_1 \text{ fresh}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma[f \mapsto \alpha_x \to \alpha_1][x \mapsto \alpha_x], e_b)$$
$$C_1' = \mathit{widen}(C_1)$$
$$\theta_2 = \mathcal{U}(\hat{\tau}_1, \theta_1\alpha_1)$$

$$\mathcal{W}(\Gamma, e_1 e_2) = \langle \theta_3\alpha_2,\ \theta_3 \circ \theta_2 \circ \theta_1, \tag{6}$$
$$(\theta_3 \circ \theta_2)C_1 \cup \theta_3 C_2 \cup \{\, \theta_3\hat{\tau}_2 <: \theta_3\alpha_1 \,\} \rangle \text{ where}$$
$$\alpha_1, \alpha_2 \text{ fresh}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)$$
$$\theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1,\ \alpha_1 \to \alpha_2)$$

$$\mathcal{W}(\Gamma, \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e) = \langle \theta_4\alpha,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \tag{7}$$
$$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2 \cup \theta_4 C_3$$
$$\cup \{\, (\theta_4 \circ \theta_3)\hat{\tau}_2 <: \alpha,\ \theta_4\hat{\tau}_3 <: \alpha \,\} \rangle$$
$$\text{where } \alpha \text{ fresh}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_c)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_t)$$
$$\langle \hat{\tau}_3, \theta_3, C_3 \rangle = \mathcal{W}((\theta_2 \circ \theta_1)\Gamma, e_e)$$
$$\theta_4 = \mathcal{U}((\theta_3 \circ \theta_2)\hat{\tau}_1, \mathit{bool})$$

$$\mathcal{W}(\Gamma, \mathbf{let}\ x = e_x\ \mathbf{in}\ e_b) = \langle \hat{\tau}_2,\ \theta_2 \circ \theta_1,\ \theta_2 C_g \cup C_2 \rangle \tag{8}$$
$$\text{where}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_x)$$
$$\Gamma' = \theta_1\Gamma$$
$$\langle \sigma_1, C_g \rangle = \mathit{generalize}(\hat{\tau}_1, \Gamma', C_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\Gamma'[x \mapsto \sigma_1], e_b)$$

$$\mathcal{W}(\Gamma, e_1 \odot e_2) = \langle \tau_\odot,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \tag{9}$$
$$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2 \rangle \text{ where}$$
$$\langle \tau_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)$$
$$\langle \tau_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)$$
$$\theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_\odot^1)$$
$$\theta_4 = \mathcal{U}(\theta_3\tau_2, \tau_\odot^2)$$
such that $\tau_\odot, \tau_\odot^1, \tau_\odot^2$ match the respective operator:
$$\tau_+ = \tau_+^1 = \tau_+^2 = \mathit{int}$$
$$\tau_< = \mathit{bool}, \tau_<^1 = \tau_<^2 = \mathit{int}$$
and similar for the other binary operators . . .

Figure 6. The general purpose part of algorithm $\mathcal{W}$.

$$\mathcal{W}(\Gamma, \mathbf{use}\ [k]\ ) = \langle \alpha \to \mathit{task}\,\beta\,\alpha, [\,], \{\, \beta \sqsupseteq k \,\} \rangle \tag{10}$$
$$\text{where } \alpha, \beta \text{ fresh}$$

$$\mathcal{W}(\Gamma, e_1 \ggg\!= e_2) = \langle \mathit{task}\,\beta\,((\theta_4 \circ \theta_3)\alpha_2), \tag{11}$$
$$\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,$$
$$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2$$
$$\cup \{\, \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) \boxplus (\theta_4\beta_2) \,\} \rangle$$
$$\text{where}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)$$
$$\beta, \beta_1, \beta_2, \alpha_1, \alpha_2 \text{ fresh}$$
$$\theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1,\ \mathit{task}\,\beta_1\,\alpha_1)$$
$$\theta_4 = \mathcal{U}(\theta_3\hat{\tau}_2,\ (\theta_3\alpha_1) \to \mathit{task}\,\beta_2\,\alpha_2)$$

$$\mathcal{W}(\Gamma, e_1\ \&\ e_2) = \langle \mathit{task}\,\beta\,((\theta_4 \circ \theta_3)\alpha_2), \tag{12}$$
$$\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,$$
$$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2$$
$$\cup \{\, \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) + (\theta_4\beta_2) \,\} \rangle$$
$$\text{where}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)$$
$$\beta, \beta_1, \beta_2, \alpha_1, \alpha_2 \text{ fresh}$$
$$\theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1, \mathit{task}\,\beta_1\,\alpha_1)$$
$$\theta_4 = \mathcal{U}(\theta_3\hat{\tau}_2, \mathit{task}\,\beta_2\,\alpha_2)$$

Figure 7. The domain specific part of algorithm $\mathcal{W}$.

constraints, unresolved subsumption constraints, freshly generated subsumption constraints, and a substitution. The effect constraints are the actual output of the solving process. They are collected and later passed on to the effect constraint solver.

Unresolved constraints are constraints about which the current pass does not yet have enough information. We do have to remember them, because as subsequent passes solve more constraints, and possibly learn more about the type variables involved, solving them might become possible.

Freshly generated subsumptions have been generated in the current pass. If there are any of those, another pass is required.

A substitution is generated if a pass uses unification to solve a constraint. The presence of a substitution indicates that a pass learned more about some type variables. Whenever a pass returns a substitution, the substitution must be applied to all unresolved constraints and another pass is required.

The function *solveSubsumption* handles constrains as follows. Clause **(1)** just filters out effect constraints. It is needed because we mix effect and subsumption constraints in the same set.

Clause **(2)** handles constraints involving two type variables. They go immediately to the set of unresolved constraints.

Clause **(3)** handles constraints involving two task types, according to the subtyping rules. Subtyping for tasks $\mathit{task}\,\beta_1\,\hat{\tau}_1 <: \mathit{task}\,\beta_2\,\hat{\tau}_2$ requires that the cost for the right task is above the cost of the left task: $\beta_2 \sqsupseteq \beta_1$, and that the types of the task values are again in the subtyping relation $\hat{\tau}_1 <: \hat{\tau}_2$.

Clause **(4)** handles constraints involving two function types. It implements co- and contravariance of function types.

Clauses **(5)**, **(6)**, **(7)** and **(8)** handle constraints where one of the types is a type variable and the other a type constructor. In all of these cases the solver learns that a variable must be a task or a function, respectively, and records this fact by generating a new constraint involving some fresh type variables.

Clause **(9)** is the catch-all clause. It handles all other constraints, in particular those involving base types, and those that cause type errors.

4.3 Effect Constraint Solving

The effect constraint solver is a translation to functional code of the worklist algorithm found in chapter 6 in Nielson et al. (1999). The

$$\begin{array}{ll}
\mathit{solveSubsumption}(\beta \sqsupseteq \varphi) = \langle \{\, \beta \sqsupseteq \varphi \,\},\ \emptyset,\ \emptyset,\ [\,]\rangle & (1)\\
\mathit{solveSubsumption}(\alpha_1 <: \alpha_2) = \langle \emptyset,\ \{\, \alpha_1 <: \alpha_2 \,\},\ \emptyset,\ [\,]\rangle & (2)\\
\mathit{solveSubsumption}(\mathit{task}\,\beta_1\,\hat{\tau}_1 <: \mathit{task}\,\beta_2\,\hat{\tau}_2) = & (3)\\
\quad \langle \{\, \beta_2 \sqsupseteq \beta_1 \,\},\ \emptyset,\ \{\, \hat{\tau}_1 <: \hat{\tau}_2 \,\},\ [\,]\rangle & \\
\mathit{solveSubsumption}(\hat{\tau}_1 \to \hat{\tau}_2 <: \hat{\tau}_3 \to \hat{\tau}_4) = & (4)\\
\quad \langle \emptyset,\ \emptyset,\ \{\, \hat{\tau}_3 <: \hat{\tau}_1, \hat{\tau}_2 <: \hat{\tau}_4 \,\},\ [\,]\rangle & \\
\mathit{solveSubsumption}(\mathit{task}\,\beta_1\,\hat{\tau}_1 <: \alpha_1) = & (5)\\
\quad \langle \emptyset,\ \emptyset,\ \{\, \mathit{task}\,\beta_1\,\hat{\tau}_1 <: \theta_1\alpha_1 \,\},\ \theta_1\rangle \text{ where} & \\
\quad \alpha_2, \beta_2 \text{ fresh} & \\
\quad \theta_1 = \mathcal{U}(\alpha_1, \mathit{task}\,\beta_2\,\alpha_2) & \\
\mathit{solveSubsumption}(\alpha_1 <: \mathit{task}\,\beta_1\,\hat{\tau}_1) = & (6)\\
\quad \langle \emptyset,\ \emptyset,\ \{\, \theta_1\alpha_1 <: \mathit{task}\,\beta_1\,\hat{\tau}_1 \,\},\ \theta_1\rangle \text{ where} & \\
\quad \alpha_2, \beta_2 \text{ fresh} & \\
\quad \theta_1 = \mathcal{U}(\alpha_1, \mathit{task}\,\beta_2\,\alpha_2) & \\
\mathit{solveSubsumption}(\hat{\tau}_1 \to \hat{\tau}_2 <: \alpha_1) = & (7)\\
\quad \langle \emptyset,\ \emptyset,\ \{\, \hat{\tau}_1 \to \hat{\tau}_2 <: \theta_1\alpha_1 \,\},\ \theta_1\rangle \text{ where} & \\
\quad \alpha_2, \alpha_3 \text{ fresh} & \\
\quad \theta_1 = \mathcal{U}(\alpha_1, \alpha_2 \to \alpha_3) & \\
\mathit{solveSubsumption}(\alpha_1 <: \hat{\tau}_1 \to \hat{\tau}_2) = & (8)\\
\quad \langle \emptyset,\ \emptyset,\ \{\, \theta_1\alpha_1 <: \hat{\tau}_1 \to \hat{\tau}_2 \,\},\ \theta_1\rangle \text{ where} & \\
\quad \alpha_2, \alpha_3 \text{ fresh} & \\
\quad \theta_1 = \mathcal{U}(\alpha_1, \alpha_2 \to \alpha_3) & \\
\mathit{solveSubsumption}(\hat{\tau}_1 <: \hat{\tau}_2) = \langle \emptyset,\ \emptyset,\ \emptyset,\ \mathcal{U}(\hat{\tau}_1, \hat{\tau}_2)\rangle & (9)
\end{array}$$

Figure 8. The subsumption constraint solver.

solver is given in Figure 9. We use set notation for the worklist for brevity.

The solver takes as input the set of effect constraints generated by the analysis so far. The output is a mapping from annotation variables to costs that satisfies the input constraints. Such a mapping is called a *solution*.

In line **(1)** the function solve starts the iteration with initial parameters. The initial worklist is the set of constraints, which means every constraint is examined at least once.

The initial solution, line **(2)**, maps every annotation variable to bottom. The mapping $\mathit{influences}$, line **(3)**, maps every annotation variable β to the set of constraints where β occurs on the right hand side, that is the set of all constraints influenced by β.

Line **(4)** defines the base case of the iteration. If the worklist is empty, the algorithm terminates.

Line **(5)** defines the case where the worklist consists of at least one constraint $\beta \sqsupseteq \varphi$ and some rest. The new cost of β is calculated in line **(6)** by taking the least upper bound of the cost so-far $s(\beta)$ and the evaluation of the annotation φ under s. The lub with $s(\beta)$ is needed because there may be several constraints with β on the left hand side, and this way the solver keeps track of the most expensive one. The flag dirty indicates whether β has gotten an increased cost, in which case all constraints that depend on β must be re-evaluated.

The worklist for the next iteration, line **(7)**, is extended, if necessary, by all the constraints that β influences. The solution for the next iteration is updated with the new cost for β.

$$\begin{array}{ll}
\mathit{solve}(C) = \mathit{iterate}(\mathit{influences},\ C,\ \mathit{initSolution}) & (1)\\
\quad \text{where} & \\
\quad \mathit{initSolution} = [\,\beta \mapsto \bot \mid \beta \in \mathit{FAV}(C)\,] & (2)\\
\quad \mathit{influences} = [\,\beta \mapsto \mathit{infl}'(\beta) \mid \beta \in \mathit{FAV}(C)\,] & (3)\\
\quad \mathit{infl}'(\beta) = \{\, \beta_1 \sqsupseteq \varphi \mid \beta_1 \sqsupseteq \varphi \in C, \beta \in \mathit{FAV}(\varphi)\,\} & \\
\mathit{iterate}(\_,\ \emptyset,\ s) = s & (4)\\
\mathit{iterate}(\mathit{influences},\ \{\, \beta \sqsupseteq \varphi \,\} \cup \mathit{rest},\ s) = & (5)\\
\quad \mathit{iterate}(\mathit{influences},\ \mathit{worklist}',\ s') \text{ where} & \\
\quad \mathit{new}\beta = [\![\varphi]\!]s \sqcup s(\beta) & (6)\\
\quad \mathit{dirty} = \mathit{new}\beta \sqsupset s(\beta) & \\
\quad \mathit{worklist}' = \begin{cases} \mathit{rest} \cup \mathit{influences}(\beta) & \text{if } \mathit{dirty}\\ \mathit{rest} & \text{otherwise}\end{cases} & (7)\\
\quad s' = s[\beta \mapsto \mathit{new}\beta] &
\end{array}$$

Figure 9. The effect constraint solver.

5. Discussion

In this section we look at example programs, discuss which challenges they pose to the analysis and explain why our method can or cannot deal with them. The employment of techniques such as subtyping and polyvariance to reduce poisoning is state-of-the-art in program analysis, see for example Gedell et al. (2006). In this section we explain how our system makes use of these techniques to increase the precision of the analysis.

A notational abbreviation. In the following examples we talk about programs of type *task*. To express the cost of a task, the analysis generates a type with annotation variable like *task* β *int* and some constraint like $\beta \sqsupseteq [3C]$. Together these indicate that the task costs $3C$ to execute. When discussing example programs it is cumbersome to explicitly mention constraints. We take the liberty to put costs directly in task types and write *task* $[3C]$ *int*.

5.1 Good Examples

Let us first look at some example programs that our analysis can deal with nicely. The subsequently described features all solve different kinds of problems, but often their uses overlap. This means there are programs that can be analyzed precisely by more than one of them. Nonetheless, there are corner cases that can only be solved by one of them.

Subtyping. There are two precursors to proper subtyping. A simple form of subtyping, called *creation-site subeffecting*, is necessary for type and effect systems to be conservative extensions of their underlying type systems. Being a conservative extension means that all programs typeable in the underlying type system can be analyzed. Creation-site subeffecting in our system means that tasks with cost k can always be considered to cost more. The prime example where this is necessary is the typing rule for conditionals. The underlying type system requires the types of the branches to be identical, which means that their costs must be identical. Creation-site subeffecting allows the cost of the cheaper task to be increased to match the cost of the more expensive one.

Creation-site subeffecting while sound, however, leads to poor results in certain situations. This effect is called *poisoning*, and happens when the same task t is used in different contexts. The cost of t is increased to match the requirements of the most expensive context, raising its cost in the other contexts as well, where it could be taken as cheaper.

Example 5.1. Consider the following program in a system without polymorphism but with creation-site subeffecting.

$$\begin{aligned}&\textbf{let}\ t = \textbf{use}\ [1C]\ 0\ \textbf{in}\\ &\textbf{let}\ s = \textbf{use}\ [3C]\ 0\ \textbf{in}\\ &\textbf{let}\ u = \textbf{if True then}\ s\ \textbf{else}\ t\ \textbf{in}\\ &t\end{aligned}$$

The conditional in the unused u forces the costs of the defining expressions of s and t to be identical, $3C$. The overall expression evaluates to just t, which actually costs $1C$ but is analyzed as costing $3C$. □

To alleviate this problem, one can allow the costs of task expressions to be adjusted not at the places where they are defined, but where they are used. This technique is called *use-site subeffecting*, and allows the cost of a task to be adjusted in each context individually. In the above example this would mean the cost of t only increases in its use in the conditional, while the use in the main expression is unaffected.

Use-site subeffecting is achieved in a type system by moving the subeffect conditions from the axioms for constants to all the rules where unification happens.

Subeffecting only applies to the annotation of the top-level type constructor of a type. It does not apply to annotations deeper in compound types. In our case such annotations occur in functions that have tasks as arguments or return values, or tasks that return tasks as values. This is where subtyping comes in. Subtyping allows subeffecting at any place in a compound type, which is important for lambda-bound higher-order functions and higher-order tasks.

Let-polymorphism. Our system features polymorphism in the usual Hindley-Milner style. When the type of a let-bound variable is not fully determined, remaining type variables are generalized. When the variable is used, the type variables get instantiated to match the type required by the context. This way the same identifier can have many types that do not influence each other.

Example 5.2. The classical example for polymorphism is the identity function. Consider the following program in our system.

$$\begin{aligned}&\textbf{let}\ id = \textbf{fn}\,x.x\ \textbf{in}\\ &id(\textbf{use}\ [1C]\ (id\,0))\end{aligned}$$

The function id has type $\forall\alpha.\alpha \to \alpha$, which can be instantiated differently in the body of the let. The outer occurrence of id gets type $\textit{task}\,[1C]\,\textit{int} \to \textit{task}\,[1C]\,\textit{int}$ and the inner occurrence $\textit{int} \to \textit{int}$. □

Let-polyvariance. Polyvariance is needed when the types of let-bound identifiers are not general enough for polymorphism to apply. Polyvariance is similar to polymorphism but binds annotation variables instead of type variables.

Example 5.3. In the following example, the function $twoTimes$ runs a given task two times in sequence. In the body of the let it is applied to tasks with different costs.

$$\begin{aligned}&\textbf{let}\ twoTimes = \textbf{fn}\,x.x \gg x\ \textbf{in}\\ &twoTimes\,(\textbf{use}\ [2C]\ 0) \gg twoTimes\,(\textbf{use}\ [1C]\ 0)\end{aligned}$$

In order to get the expected analysis result, $6C$, $twoTimes$ needs the following types in the first and second call respectively.

$$\textit{task}\,[2C]\,\textit{int} \to \textit{task}\,[4C]\,\textit{int} \tag{1}$$
$$\textit{task}\,[1C]\,\textit{int} \to \textit{task}\,[2C]\,\textit{int} \tag{2}$$

Polymorphism does not help because the sequence operator in the definition of $twoTimes$ forces its type to be a function from tasks to tasks, hence there is no type variable that can be instantiated to task types with different costs.

Without polyvariance the type system must give $twoTimes$ the more expensive type (1) for both calls, leading to a result of $8C$.

With polyvariance the type system can assign the following type to $twoTimes$.

$$\forall\beta.\textit{task}\,\beta\,\textit{int} \to \textit{task}\,(\beta \boxplus \beta)\,\textit{int}$$

This type can be instantiated to the types (1) and (2) above. □

Example 5.4. In Example 5.1 we argued that subtyping prevents poisoning. In fact, because the identifiers in that example are let-bound, polyvariance also prevents poisoning. In the following program however, polyvariance does not apply because s and t are lambda-bound instead of let-bound.

$$\begin{aligned}&(\textbf{fn}\ t.\textbf{fn}\ s.const\ t\ (\textbf{if True then}\ s\ \textbf{else}\ t))\\ &\quad(\textbf{use}\ [1C]\ 0)(\textbf{use}\ [3C]\ 0)\end{aligned}$$

Only subtyping can prevent poisoning so that the expression has cost $1C$. □

Recursion. To analyze recursive functions, the algorithm makes use of a technique called *widening*. Widening solves the problem that recursion depth, or termination for that matter, is Turing complete. A recursive function that uses some resource in each iteration could potentially require an infinite amount of resources. A naive implementation using constraints leads to recursive constraints, that is, a group of constraints where an annotation variable occurs on both the left and the right-hand side of constraints. In the simplest case this happens in the same constraint, for example:

$$\{\,\beta \sqsupseteq \beta \boxplus \beta, \beta \sqsupseteq [1C]\,\}.$$

This constraint set has a solution in our domain, namely $[\beta \mapsto \infty]$. Kleene-style fixpoint iteration however cannot compute this solution, because starting at $[\beta \mapsto \bot]$ the iteration diverges.

Widening guarantees that fixpoint iteration terminates in the presence of recursive constraints. The widening operator ∇, formally defined in Definition 2.10, takes two arguments and yields infinity if the right argument is above the left one.

Our algorithm applies widening to all constraints coming from the bodies of recursive functions. See Definition 4.1. The above constraint set becomes:

$$\{\,\beta \sqsupseteq \beta \mathbin{\nabla} \beta \boxplus \beta, \beta \sqsupseteq \beta \mathbin{\nabla} [1C]\,\}.$$

This causes the fixpoint iteration to stepwise calculate the following values for β. After the third step, the value for β no longer changes and the process terminates.

$$\begin{aligned}&\beta = \bot\\ &\beta = 1C\\ &\beta = 1C \mathbin{\nabla} (1C \boxplus 1C) = 1C \mathbin{\nabla} 2C = \infty C\end{aligned}$$

Example 5.5. The following program is an infinite recursion where each iteration costs one unit of a consumable resource C and one unit of a reusable resource R. The algorithm estimates the cost of the program as expected with $\infty C + 1R$.

$$(\textbf{fix}\,f x.x \gg (f x))(\textbf{use}\ [1C + 1R]\ 0)$$ □

Example 5.6. In the following program, the function doubles its argument in each recursive call. The parameter costs $1R$, but reusables cannot be shared between parallel tasks. The cost of each iteration is twice the cost of the previous one. Our algorithm estimates the cost of the program with ∞R.

$$(\textbf{fix}\,f x.x \gg (f(x \,\&\, x)))(\textbf{use}\ [1R]\ 0)$$ □

Higher-order tasks. Tasks not only consume resources, they also produce values. These values can be of any type, in particular they can be tasks. This is useful because in the world of iTasks, managing tasks is a task. Think about a person deciding which task to execute. The task of making the decision itself has a cost, as does the resulting task. Another example could be that the procedure of investigating the nature of a fire has as result the corresponding firefighting task.

Example 5.7. In the following program, *decide* is a task whose result is a task. The expression c is some condition whose details are not important.

$$\begin{aligned}&\mathbf{let}\ \mathit{decide} = \mathbf{use}\ [1C]\ (\mathbf{if}\ c\ \mathbf{then}\\&\quad \mathbf{use}\ [3R]\ 0\ \mathbf{else}\ \mathbf{use}\ [4R]\ 0)\ \mathbf{in}\\&\mathbf{let}\ \mathit{execute} = \mathit{id}\ \mathbf{in}\\&\mathit{decide} \ggg\!= \mathit{execute}\end{aligned}$$

The task *decide* has type *task* $[1C]$ (*task* $[4R]$ *int*). The overall program has predicted cost $1C + 4R$, because first the decision is made and then the resulting task is executed. ☐

5.2 Challenging Examples

The analysis always gives safe approximations, which means the cost of actually running a program is never higher than what the analysis predicts. In the examples so far the predicted costs match what a programmer would reasonably expect by inspecting the programs. In the following examples we look at programs where the analysis gives results that are unexpected unless the programmer is familiar with the specifics of the analysis algorithm.

Widening. The widening operator yields infinity immediately when its right-hand side is above the left-hand side. This is a quite aggressive strategy which gives bad results for constraint sets where iteration would otherwise stabilize after a finite number of steps.

Example 5.8. The following program uses the fixpoint combinator but has no recursive calls. This causes the constraints from the function body to be unnecessarily subjected to widening. Solving the constraints involves an intermediate solution where one of the widened constraints goes to infinity, whereas without widening, iteration would stabilize at a finite value after a couple of rounds.

$$(\mathbf{fix}\ f x.\mathbf{if}\ \mathbf{True}\ \mathbf{then}\ x\ \mathbf{else}\ (x \gg x))(\mathbf{use}\ [1C]\ 0)$$

The analysis outcome we would like to have is $2C$ because that is the worst case of the conditional. The prediction our algorithm gives is ∞C, for the reasons described above. ☐

Lambda-bound functions. Polymorphism and polyvariance only apply to let-bound variables. Lambda-bound variables are still subject to subtyping, which allows good results if they are used as arguments in function calls. If a lambda-bound variable is used in the function position in a function call, neither polymorphism nor subtyping applies, which can result in poisoning.

Example 5.9. In the following example, the identity function is lambda-bound to f. There are two applications of f, of which the second is ignored.

$$\begin{aligned}&(\mathbf{fn}\ f.\mathit{const}\ (f(\mathbf{use}\ [1C]\ 0))\\&\qquad\qquad (f(\mathbf{use}\ [3C]\ 0)))(\mathbf{fn}\ x.x)\end{aligned}$$

The second application nonetheless contributes to the type of f:

$$\mathit{task}\ [3C]\ \mathit{int} \to \mathit{task}\ [3C]\ \mathit{int}$$

The analysis result of this program is $3C$, whereas a programmer might expect $1C$. ☐

There are more problems with lambda-bound functions. The inability to give different types to different uses of a function can cause non-termination of the fixpoint iteration.

Example 5.10. In the following program, the argument f of *twice* is applied to its own result. In our current implementation, the worklist algorithm diverges when trying to analyze this program. The actual cost when running the program is $[4C]$.

$$\begin{aligned}&\mathbf{let}\ \mathit{twice} = \mathbf{fn}\ f.\mathbf{fn}\ x.f(f\ x)\ \mathbf{in}\\&\mathbf{let}\ g = \mathbf{fn}\ t.t \gg t\ \mathbf{in}\\&\mathit{twice}\ g\ (\mathbf{use}\ [1C]\ 0)\end{aligned}$$

Let us hand-wave our way through the type inference algorithm to understand the cause of the problem. f is a function, so it must have type $\alpha_1 \to \alpha_2$. Furthermore, f is applied to its own result, so [t-app] generates a subtype constraint $\alpha_2 <: \alpha_1$. The fact that *twice* is let-bound and therefore α_1 and α_2 are generalized does not help, because the problem emerges when the type and the constraint of *twice* get instantiated in the second let body. In the second let body, g has type *task* β *int* $\to$ *task* $(\beta \boxplus \beta)$ *int*, with constraint $\beta \sqsupseteq [1C]$. Instantiating the type of *twice* for g, we get $\alpha_1 =$ *task* β *int* and $\alpha_2 =$ *task* $(\beta \boxplus \beta)$ *int*. Instantiating the constraint of *twice* for g, we get *task* $(\beta \boxplus \beta)$ *int* $<:$ *task* β *int*. According to the rules of subtyping this gives the constraint $\beta \sqsupseteq \beta \boxplus \beta$.

In the actual analysis algorithm, there are many more intermediate type- and annotation variables with trivial constraints like $\beta_7 \sqsupseteq \beta_8$ that connect them. The details do not matter. What matters is that the resulting constraint set essentially looks like this:

$$\{\, \beta \sqsupseteq [1C],\ \beta \sqsupseteq \beta \boxplus \beta \,\}$$

This constraint set has a solution in $\mathbb{D}$, namely $[\beta \mapsto \infty]$. This solution cannot be computed by fixpoint iteration.

The problem here is not the absence of widening. In fact applying widening, say after some fixed number of iterations, would lead to overly pessimistic results in other cases. These are cases where non-recursive constraints would reach a stable solution after many iterations. Example 5.8 demonstrates what can happen when widening is applied unnecessarily.

The real problem here is the emergence of recursive constraints in the first place. Because f is lambda-bound, it gets the same type in both occurrences in $f(f\ x)$. The problem would not occur if we were able to give f a polymorphic type. This would require higher-ranked polymorphism, which could be a topic for future work. With higher-ranked polymorphism, the resulting constraint set would look as follows.

$$\{\, \beta_1 \sqsupseteq [1C],\ \beta_2 \sqsupseteq \beta_1 \boxplus \beta_1,\ \beta_3 \sqsupseteq \beta_2 \boxplus \beta_2 \,\}$$

The second and third constraint come from the different instantiations of f's type. The least solution to this constraint set would have $\beta_3 \mapsto [4C]$, which is the result a programmer would expect for the program in this example. ☐

6. Future Work

There are three directions in which we would like to extend the system. They all revolve around making the analysis accessible to iTasks programmers.

First, we need a solution to the non-termination issue of Example 5.10. A pragmatic approach, which is acceptable to programmers, as various discussions have suggested, is artificially limiting the height of the lattice $\mathbb{D}$. This can be done by parameterizing the analysis with a finite upper bound for each resource. Fixpoint iteration terminates when this upper bound is reached, which always happens in finitely many steps. In the real world any resource is only available in a limited quantity anyway, otherwise cost analysis would be pointless. For a programmer it conveys the same information whether a task uses $n + 1$ or infinite units of a resource of which there are only n units.

Limiting the height of the domain furthermore allows us to remove widening altogether, which also solves the problem that widening unnecessarily overestimates costs in situations like Example 5.8.

Second, we would like to make the analysis applicable to real iTasks programs. As it stands, the analysis takes as input programs written in the language described in this paper. The language has been designed to be a minimal variant of Clean and iTasks, but there are some features missing that are essential to everyday functional programming, most notably algebraic data types, pattern matching, and mutual recursion.

Part of this second point is integration with Tonic (Stutterheim et al. 2014), the system that can visualize iTasks programs and inspect them at run time. In particular, we would like Tonic to display predicted costs in static blueprints and actual costs in dynamic blueprints.

The third direction needed for real-world application is a concept for error reporting. The current implementation reports the overall cost of a program, but a programmer needs more information when that cost exceeds the available limit. This information is not as simple as pointing to an ill-typed expression in the program, because if there are many tasks whose costs together exceed the limit, there is not one obvious culprit. There is a technique called error slicing which identifies all parts of a program that contribute to an error message. Type error slicing (Haack and Wells 2004) looks promising for our purpose from what we have seen so far, but further study is needed.

7. Related Work

For readers familiar with program analysis, and in particular type and effect systems, it should be obvious that we are following in the footsteps of Nielson and Nielson and their various coauthors. In particular, there is the textbook on program analysis (Nielson et al. 1999), and their triptych on polymorphic subtyping with Amtoft (Amtoft et al. 1997; Nielson et al. 1996b,a).

The paper on the IO monad in Haskell (Peyton-Jones 2001) served as inspiration for various aspects of our dynamic semantics.

Other papers whose approach to type and effect systems influenced our work are Gedell et al. (2006), the work on exception analysis by Koot and Hage (2015), the usage analysis by Hage et al. (2007), and the security analysis by Weijers et al. (2014).

The distinction between consumable and reusable resources is common practice in the field of artificial intelligence and automated planning (Ghallab et al. 2004).

Papers that deal with the analysis of resource consumption of programs often focus on computational resources of the program itself like memory usage and execution time. Notable examples are Vasconcelos and Hammond (2003) and Jost et al. (2010).

Kersten et al. (2014) have a resource analysis similar in spirit to ours. They focus on a single resource however, energy consumption of hardware components, and their language is imperative with first-order functions.

The programming language Clean and the iTasks system, for which our analysis is ultimately designed, are described in the Clean language report (Plasmeijer and van Eekelen 2002) and the paper by Plasmeijer et al. (2011).

Acknowledgments

We would like to thank Ruud Koot, Jurriën Stutterheim, Tim Steenvoorden, Terry Stroup, Chris Elings, Fok Bolderheij, Marko van Eekelen, and Adelbert Bronkhorst for many hours of fruitful discussion.

This research is funded by the Royal Netherlands Navy and TNO.

References

T. Amtoft, F. Nielson, H. R. Nielson, and J. Ammann. Polymorphic subtyping for effect analysis: The dynamic semantics. *Lecture Notes in Computer Science*, 1192, 1997.

T. Gedell, J. Gustavsson, and J. Svenningsson. Polymorphism, subtyping, whole program analysis and accurate data types in usage analysis. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia*, pages 200–216. Springer, 2006.

M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Haack and Wells. Type error slicing in implicitly typed higher-order languages. *SCIPROG: Science of Computer Programming*, 50, 2004.

J. Hage, S. Holdermans, and A. Middelkoop. A generic usage analysis with subeffect qualifiers. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 235–246. ACM, 2007.

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. *ACM SIGPLAN Notices*, 45(1):223–236, Jan. 2010.

R. Kersten, P. Parisen Toldin, B. van Gastel, and M. van Eekelen. A Hoare logic for energy consumption analysis. In *Proceedings of the Third International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'13)*, volume 8552 of *LNCS*, pages 93–109. Springer, 2014.

R. Koot and J. Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In K. Asai and K. Sagonas, editors, *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*, pages 127–138. ACM, 2015.

F. Nielson, H. R. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The algorithm. In M. Dam, editor, *LOMAPS*, volume 1192 of *Lecture Notes in Computer Science*, pages 207–243. Springer, 1996a.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In M. Dam, editor, *LOMAPS*, volume 1192 of *Lecture Notes in Computer Science*, pages 141–171. Springer, 1996b.

Object Management Group. Business process model and notation (BPMN) version 1.2. Technical report, Object Management Group, 2009.

S. Peyton-Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In C. A. R. Hoare, M. Broy, and R. Steinbrueggen, editors, *Engineering Theories of Software Construction*, NATO ASI Series, pages 47–96. IOS Press, 2001. Marktoberdorf Summer School 2000.

R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. van Noort, and J. van Groningen. iTasks for a change: Type-safe run-time change in dynamically evolving workflows. In S. Khoo and J. Siek, editors, *Proceedings of the PEPM '11, Austin, TX, USA*, pages 151–160. ACM Press, 2011.

J. Stutterheim, R. Plasmeijer, and P. Achten. Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, volume 8843 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 2014.

P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. W. Trinder, G. Michaelson, and R. Pena, editors, *IFL*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2003.

J. Weijers, J. Hage, and S. Holdermans. Security type error diagnosis for higher-order, polymorphic languages. *Sci. Comput. Program*, 95:200–218, 2014.