The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines

Markus Klinik
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
m.klinik@cs.ru.nl

Jan Martin Jansen
Netherlands Defence Academy (NLDA)
Den Helder, The Netherlands
jm.jansen.04@mindef.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

In this paper we present a static analysis for costs of higher-order workflows, where costs are maps from resource types to simple functions over time. We present a type and effect system together with an algorithm that yields safe approximations for the cost functions of programs.

CCS CONCEPTS

• **Theory of computation → Program analysis**;

KEYWORDS

workflow systems, resource modelling, type and effect systems

ACM Reference Format:
Markus Klinik, Jan Martin Jansen, and Rinus Plasmeijer. 2017. The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205368.3205376

1 INTRODUCTION

Task Oriented Programming (TOP) is a programming paradigm that allows specifying workflows in a declarative way. TOP programs are distributed applications where users work together on the internet. They are built using the four concepts *tasks*, *shared data*, *generic interaction*, and *task combinators*. There is an implementation of TOP called iTasks [Plasmeijer et al. 2012], which comes as an embedded domain specific language in the pure, lazy functional programming language Clean. A demonstration of the capabilities of iTasks can be found in Lijnse et al. [2012], where it has been used to write a crisis management tool for the Dutch coast guard.

In this paper we focus on tasks and task combinators. We are interested in statically analysing a simplified variant of TOP programs, enriched with costs and durations for basic tasks. In previous work [Klinik et al. 2017] we presented a static analysis for workflows that, given costs for basic tasks, yields a safe approximation of the cost of a whole program. Costs are maps from resource types to numbers, one number for each type of resource a program requires. In this paper we refine the notion of cost by including time, in the sense that each basic task gets a duration, and its cost refers to this duration. The result of the extended analysis is not a single number, but a function over time. The main contribution of this paper is the development of a two-dimensional cost model suitable for both the dynamic and static semantics (Section 2). We also present a type system (Section 3) and an implementation (Section 4) for the analysis.

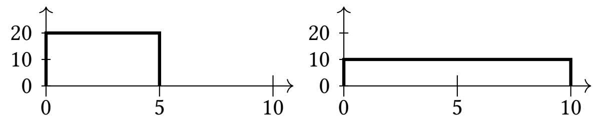


Figure 1: Cost skylines for a birthday and a wedding

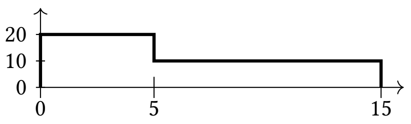


Figure 2: Cost skyline for a birthday *followed by* a wedding

1.1 Basic Ideas

We would like to represent the cost of executing tasks by functions over time. Complex tasks are composed of simpler ones, and so should be their cost functions. By combining cost functions of simple tasks using operations that correspond to task composition, we obtain cost functions for complex tasks.

Example 1.1. Consider the two tasks of hosting a birthday party and a wedding, which both require chairs. Let's say the birthday party requires 20 chairs and takes five hours, while the wedding requires ten chairs and takes ten hours. We visualize these costs over time in diagrams called *skylines*. The skylines are shown in Figure 1.

Hosting the birthday party first and immediately after it the wedding requires being able to supply 20 chairs for the first five hours and 10 chairs for the hours 5 to 15. The corresponding skyline is shown in Figure 2.

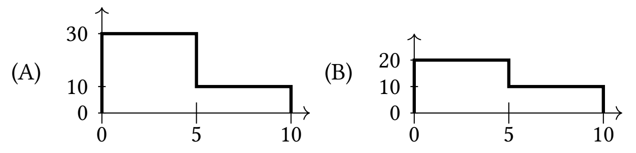


Figure 3: (A) Cost skyline for a birthday *and* a wedding. (B) Cost skyline for a birthday *or* a wedding

Hosting the birthday party and the wedding at the same time requires 30 chairs for the first five hours, and 10 chairs for the remaining five hours. See Figure 3 (A).

If, for whatever reason, either the wedding or the birthday takes place, then actually either 20 chairs for five hours or 10 chairs for ten hours are used. In order to be prepared for both situations, a host must calculate with 20 chairs for five hours and ten chairs for another five hours. See Figure 3 (B).

In this example, we considered chairs, which are a reusable resource. Combining skylines of consumable resources works differently, because those can only increase and never decrease.

2 SYNTAX AND SEMANTICS

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

We consider two kinds of resources, *consumables* and *reusables*. Consumable resources are used up when a task that requires them is executed. Reusable resources are claimed exclusively during execution of a task and become available again upon completion.

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.

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

The general-purpose part of the language has Boolean and integer constants b and i, the unit value $()$, 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.

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, and t its duration. Costs are given in a polynomial-like syntax where n is a natural number and u the unit of a resource. Durations t are given as non-zero natural numbers. For example the expression "**use** $[2S + 3B, 20]$ 5" may denote a task that uses 2 screwdrivers, 3 bottles of wine, takes 20 minutes to execute, and yields the value 5. Costs, time, 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.

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 $\gg$ and $\ggg=$ have identical cost behavior, we ignore $\ggg=$ in the formal parts of this paper for simplicity.

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 to the language is standard procedure and can be found in any textbook on type systems. We take the liberty of bending the semantics of the parallel combinator a bit to avoid tuples, saving some space in this paper. Both arguments to (&) and its result are of type *task* ().

2.2 A Domain for Representing Costs Over Time

In this section we develop a model for costs over time, suitable for both the dynamic and static semantics.

Definition 2.1. (Extended natural numbers) The extended natural numbers are the natural numbers with infinity: $\overline{\mathbb{N}} = \{0, 1, 2, \ldots, \infty\}$. The only operations we need are addition and comparison, which are extended with ∞ in the natural way.

Definition 2.2. (Time) For our purposes, a *point* in time is an element of $\overline{\mathbb{N}}$. A *duration* is a non-zero natural number. □

Programmers can only give finite durations to basic tasks, but the analysis can give infinite durations to programs when needed.

Definition 2.3. (Skylines) The cost over time for a single resource is given by time series $\overline{\mathbb{N}} \to \overline{\mathbb{N}}$, called *skyline*. The set of all skylines is called $\mathbb{S}$. □

Definition 2.4. (Skyline ordering) Skylines are ordered pointwise. Let r, s be skylines.

$$r \sqsubseteq s \text{ iff } r(t) \sqsubseteq s(t) \text{ for all } t \in \overline{\mathbb{N}} \qquad \square$$

Definition 2.5. (Adding skylines) The sum of two skylines is defined pointwise.

$$(r + s)(t) = r(t) + s(t) \qquad \square$$

Definition 2.6. (Merging skylines) The merge of two skylines ($\sqcup$) is defined as their pointwise maximum.

$$(r \sqcup s)(t) = r(t) \sqcup s(t) \qquad \square$$

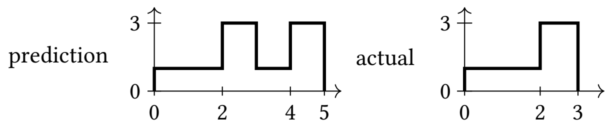


Figure 4: A prediction of the conditional (left) must predict the spike at time 2 in the actual skyline (right).

Appending skylines requires a bit more consideration. Tasks in the branches of a conditional can have different durations. For the overall skyline of a conditional, it is not sufficient to have the maximum length of the branches, as Example 2.7 illustrates.

Example 2.7. The following program has a conditional with two cheap tasks of different length, followed by an expensive task.

```
( if  True
    then use (1R,2)  0
    else  use (1R,4)  0
)  ≫ use (3R,1)  0
```

A safe prediction must take into account that the spike can happen at time 2 or at time 4, see Figure 4 □

The set of possible start times of subsequent tasks grows with nested conditionals and conditionals in sequence. Our analysis keeps track of the earliest and latest times a task can end. All operations on costs must take this interval into account. End intervals belong to costs (Definition 2.13), not to individual skylines. The end interval of a cost applies to all its skylines. When we talk about the end interval of a skyline, we mean the end interval of the cost of which the skyline is part of.

Definition 2.8. An *end interval* is a pair $[x, y] : \mathbb{N} \times \overline{\mathbb{N}}$. □

Definition 2.9. (Operations on end intervals)

$$\begin{aligned}[][x_1, y_1] \sqcup [x_2, y_2] &= [\min(x_1, x_2), \max(y_1, y_2)]\\ [x_1, y_1] + [x_2, y_2] &= [\max(x_1, x_2), \max(y_1, y_2)]\\ [x_1, y_1] \mathbin{+\!\!+} [x_2, y_2] &= [x_1 + x_2, y_1 + y_2]\end{aligned}$$ □

The operations $\sqcup, +, \mathbin{+\!\!+}$ correspond to conditionals, parallel, and sequential composition of tasks respectively.

The earliest possible end point of a *conditional* is the earliest one of the branches. The latest possible end point is the latest one. A *parallel* composition terminates when both branches terminate. The earliest possible end time of parallel tasks is therefore the maximum start point of their end intervals. The latest possible end time is the maximum end time of the tasks. The *sequential* composition of two tasks terminates after the second task terminates. The earliest possible end time is therefore the sum of the earliest end times of the operands, the latest possible end time is the sum of the latest end times of the operands.

Appending skylines needs to take end intervals into account. It makes the simplifying but safe assumption that the second skyline can start anywhere in the end interval. Appending consumable and reusable skylines works slightly differently.

Definition 2.10. The function $\text{shift}(s, n)$ shifts a skyline s to the right by n. The function $\text{bump}(s, n, x)$ shifts a skyline s to the right by n and upwards by x.

Definition 2.11. (Appending reusable skylines) To append reusable skylines r and s, we shift s to every point where it could start and merge all possibilities. Formally, let e be the end interval of r.

$$r \mathbin{+\!\!+} s = r \sqcup \bigsqcup \{ \text{shift}(s, n) \mid n \in e \}$$ □

Definition 2.12. (Appending consumable skylines) To append consumable skylines r and s, we bump s to every point where it could start and merge all possibilities. Formally, let e be the end interval of r.

$$r \mathbin{+\!\!+} s = r \sqcup \bigsqcup \{ \text{bump}(s, n, r(n)) \mid n \in e \}$$ □

Definition 2.13. (Predicted cost) The *predicted cost* γ of a program is a tuple $\langle c, e \rangle$ where $c : U \to \mathbb{S}$ is a family of skylines, one for each resource in U, and e is an end interval. The end interval must be compatible with the skylines, which means it must end where the longest skyline ends. □

Definition 2.14. (Actual cost) The *actual cost* of a program uses the same construction as the predicted cost, but the end interval collapses to a single finite point. □

Definition 2.15. (Cost ordering) The ordering on costs must respect skylines and end intervals.

$$\begin{aligned}\langle c_1, e_1 \rangle \sqsupseteq \langle c_2, e_2 \rangle \iff & e_1 \sqsupseteq e_2 \text{ and}\\ & s_1 \sqsupseteq s_2 \text{ for all skylines in } c_1, c_2.\end{aligned}$$

The ordering on end intervals is interval inclusion. □

Definition 2.16. (Operations on costs) Operations on costs $\sqcup, +, \mathbin{+\!\!+}$ are defined in terms of the respective operations on skylines and end intervals. □

2.3 Operational Semantics

We split the operational semantics in two parts. Both are small-step structural operational semantics. 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 $\gg\!=$.

The general purpose semantics relates expressions using rules of the form $e \to e'$. The domain specific semantics with rules of the form $\langle e, \gamma \rangle \twoheadrightarrow \langle e', \gamma' \rangle$ relates expressions e while recording the costs γ which 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".

The split into two semantics comes from the fact that we are dealing with two languages: a domain specific task language embedded in a functional host language. Task expressions are values for the host semantics, which ensures that resources are only consumed when the task semantics makes a step. This gives the same cost behavior for both call-by-name and call-by-value host languages. Our choice for a call-by-name host language is arbitrary, motivated by the fact that Clean is lazy. The setup is very similar to the treatment of IO in Haskell in Peyton-Jones [2001].

In order to define the semantics for parallel composition, we add a new syntactic form to the language, called *process pool*. When the parallel composition of two tasks needs to be reduced, a process pool springs into existence and keeps track of the costs of the tasks

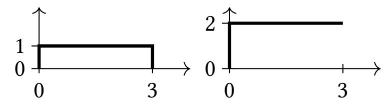


Figure 5: Basic skylines of sky$(1R + 2C, 3)$

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

$$[\text{gs-app-cong}] \quad \frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

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

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

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

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

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

Figure 6: Semantics for general purpose expressions

individually, so that no sharing of resources takes place. Process pools only exist temporarily during reduction and disappear once both tasks have been executed. Process pools in our language hold exactly two tasks, but can be nested.

$$e ::= \ldots \mid pp(e_1 \,\&\, e_2, \gamma_1, \gamma_2)$$

In a process pool $e_1 \,\&\, e_2$ are two tasks in progress of being executed and γ_1 and γ_2 are their respective costs.

Definition 2.17. (Basic skylines) The function *sky* takes a resource requirement and a duration d and returns a predicted cost with basic skylines. The end interval is the single point d. Basic skylines for each resource all have duration d and the height from their entry in the resource requirement. □

For example, the skylines of sky$(1R + 2C, 3)$ look as in Figure 5. This figure also shows how we visualize consumable skylines. They never decrease, but always stay at the height they have reached.

The general purpose semantics is given in Figure 6. The domain specific semantics is given in Figure 7.

The general purpose rules implement a call-by-name lambda calculus and should hold no surprises for readers familiar with the subject.

The domain specific rules reduce task expressions while recording resource consumption. The congruence and pure rules in the domain specific semantics specify where reductions can happen. The interesting rules are **[ds-use]**, **[ds-seq-ret]**, **[ds-par-init]**, and **[ds-par-ret]**.

The rule **[ds-use]** specifies how a basic task is evaluated: The function *sky* creates basic skylines of duration d for all resources in k, and these skylines are then appended to the costs so-far γ. **[ds-seq-ret]** sees that the left hand side has terminated and continues with the right hand side. It does not change the costs. **[ds-par-init]** creates a process pool for the two tasks, and initializes each with local costs of zero. **[ds-par-ret]** applies when both tasks in a

$$[\text{ds-use}] \quad \langle \mathbf{use}\ [k,d]\ e,\ \gamma \rangle \twoheadrightarrow \langle \mathbf{return}\ e,\ \gamma +\!\!+\ sky(k,d) \rangle$$

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

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

$$[\text{ds-seq-cong}] \quad \frac{\langle e_1, \gamma \rangle \twoheadrightarrow \langle e_1', \gamma' \rangle}{\langle e_1 \gg e_2,\ \gamma \rangle \twoheadrightarrow \langle e_1' \gg e_2,\ \gamma' \rangle}$$

$$[\text{ds-seq-pure}] \quad \frac{e_1 \rightarrow e_1'}{\langle e_1 \gg e_2, \gamma \rangle \twoheadrightarrow \langle e_1' \gg e_2, \gamma \rangle}$$

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

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

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

$$[\text{ds-par-cong-r}] \quad \text{symmetric}$$

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

$$[\text{ds-par-pure-r}] \quad \text{symmetric}$$

Figure 7: Semantics for task expressions

process pool have been fully evaluated. The costs of each task have been tracked independently in the local costs γ_1 and γ_2. The idea is that these tasks run in parallel, so they cannot re-use each others resources. The local costs are added up and appended to the cost so far.

Example 2.18. In this example we trace the reduction of a program, together with the progression of resource consumption. The program executes three tasks in sequence, where the middle task uses a different resource than the others. The end interval in the dynamic semantics always collapses to a single point, which is the elapsed time. For simplicity, we write configurations of the operational semantics as $\langle e, c, t \rangle$, where e is the expression to be reduced, c all skylines so-far, and t the single-point end interval.

$$\begin{array}{lr} & \langle \mathbf{use}\ [1C,1]\ 0 \gg \mathbf{use}\ [1R,1]\ 0 \gg \mathbf{use}\ [1C,1]\ 0, c_0, 0 \rangle \\ \twoheadrightarrow & \langle \mathbf{return}\ 0 \gg \mathbf{use}\ [1R,1]\ 0 \gg \mathbf{use}\ [1C,1]\ 0, c_1, 1 \rangle \\ \twoheadrightarrow & \langle \mathbf{use}\ [1R,1]\ 0 \gg \mathbf{use}\ [1C,1]\ 0, c_1, 1 \rangle \\ \twoheadrightarrow & \langle \mathbf{return}\ 0 \gg \mathbf{use}\ [1C,1]\ 0, c_2, 2 \rangle \\ \twoheadrightarrow & \langle \mathbf{use}\ [1C,1]\ 0, c_2, 2 \rangle \\ \twoheadrightarrow & \langle \mathbf{return}\ 0, c_3, 3 \rangle \end{array}$$

The progression of resource consumption is shown in Figure 8. First, the skyline for C takes a step (c_1). Then, the skyline for R takes a step (c_2), and finally the skyline for C takes another step (c_3).

It is important to notice that the steps taken are not appended directly at the end of the skyline of the previous step, but at the

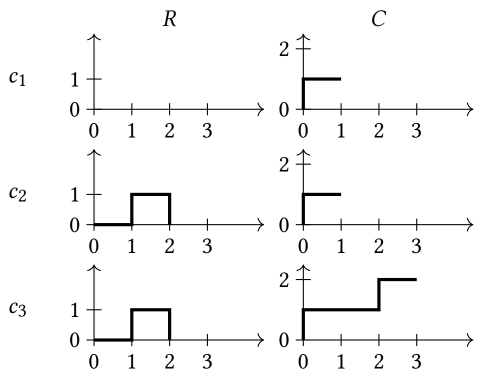


Figure 8: Progression of resource consumption of three tasks in sequence.

point in time where the task is started. The semantics knows this because it tracks the elapsed time in the end interval. That is why the skyline for C stays at length 1 in c_2, and makes its next step at time 2 in c_3. □

Example 2.19. This example covers the reduction of a program with parallel tasks, illustrating how process pools occur during execution. All three tasks in the program use the same reusable resource R.

$$\begin{aligned}
&\langle \textbf{use } [1R,1]\ 0 \gg (\textbf{use } [1R,1]\ () \mathbin{\&} \textbf{use } [1R,2]\ ()), c_0, 0\rangle && (1)\\
\twoheadrightarrow\ &\langle \textbf{return } 0 \gg (\textbf{use } [1R,1]\ () \mathbin{\&} \textbf{use } [1R,2]\ ()), c_1, 1\rangle\\
\twoheadrightarrow\ &\langle \textbf{use } [1R,1]\ () \mathbin{\&} \textbf{use } [1R,2]\ (), c_1, 1\rangle\\
\twoheadrightarrow\ &\langle pp(\textbf{use } [1R,1]\ () \mathbin{\&} \textbf{use } [1R,2]\ (), \bot, \bot), c_1, 1\rangle && (2)\\
\twoheadrightarrow\ &\langle pp(\textbf{return } () \mathbin{\&} \textbf{use } [1R,2]\ (), d_1, \bot), c_1, 1\rangle\\
\twoheadrightarrow\ &\langle pp(\textbf{return } () \mathbin{\&} \textbf{return } (), d_1, d_2), c_1, 1\rangle && (3)\\
\twoheadrightarrow\ &\langle \textbf{return } (), c_2, 3\rangle
\end{aligned}$$

The costs of this reduction are shown in Figure 9. In line (1), the first task is executed, creating the skyline c_1. In line (2), a process pool is created to prepare the execution of the parallel tasks. The process pool is initialized with empty initial costs for both tasks. In line (3), both tasks have been executed, with local skylines d_1 and d_2. In the last line, the process pool gets destroyed and the local costs get incorporated into the global cost with the formula $c_2 = c_1 \mathbin{+\!\!+} (d_1 + d_2)$.

3 STATIC SEMANTICS

We now describe the type system for skylines.

3.1 Calculating With Infinity

This section motivates the inclusion of infinite height and length of skylines. At any point during execution of a workflow, the amount of expended resources so-far is finite, as is the elapsed time. The static analysis must be able to express that a workflow might have infinite cost or might take infinite time, or both. The following examples illustrate these kind of situations.

Example 3.1. The following program starts with a constant task, then has a loop that runs n tasks in parallel, and finally ends with

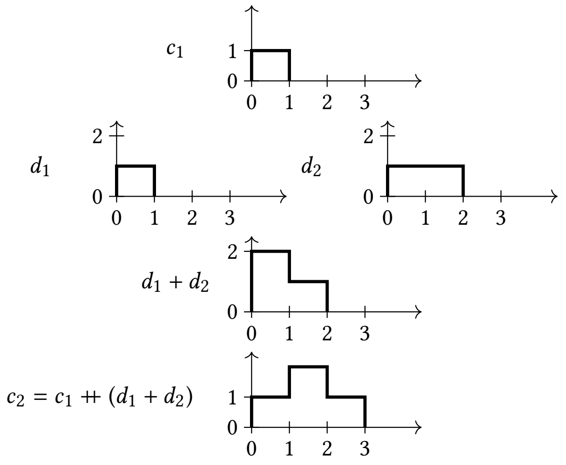


Figure 9: Progression of resource consumption of three tasks where two are in parallel.

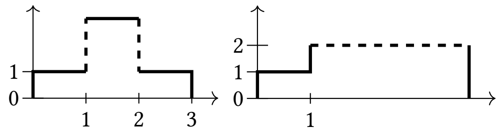


Figure 10: A skyline with infinite cost but finite length (left). A skyline with infinite length but finite cost (right). Dashed lines are infinitely long.

a constant task. The analysis cannot know how large the cost of the loop is going to be, so it must estimate it with infinity. The predicted skyline is shown in Figure 10 on the left side. The dashed part of the skyline stands for a cost of infinite height, but finite length.

```
let t = use [1R,1] 0 in
let loop = fix f n .
  if (n ≤ 1) then t else t & f (n−1) in
t ≫ loop 5 t ≫ t
```

A loop whose length can only be predicted by infinity is one that runs a parametric number of tasks in sequence. The predicted skyline of such a program is shown in Figure 10 on the right side. The dashed part stands for a cost of finite height but infinite length. □

Example 3.2. Loops that require consumable resources often need predicted skylines with both infinite height and length. Consider the following program. It runs a task requiring a consumable resource a given number of times in sequence.

```
let repeat = fix f n. fn t.
  if n ≤ 0 then t else t ≫ (f (n−1) t) in
repeat 3 (use [1C,1] 0)
```

The cost of this program cannot be statically bounded in height or in length, so the best approximation our analysis can make is a skyline that immediately jumps to infinity and stays there forever. □

3.2 Constraints

Definition 3.3. (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.

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

k stands for cost constants together with a duration, like $[2C+3R, 5]$. β stands for annotation variables. The operators $\sqcup, +, +\!\!\!+, \nabla$ stand for their respective operations on costs. □

Definition 3.4. (Constraints) Our system has two kinds of constraints, subtyping constraints and effect constraints. Constraints are formed by the following grammar.

$$c ::= \hat{\tau}_1 <: \hat{\tau}_2 \mid \beta \sqsupseteq \varphi$$

Constraints of the form $\hat{\tau}_1 <: \hat{\tau}_2$ are called *subtyping* constraints, and record the fact that $\hat{\tau}_1$ must be a subtype of $\hat{\tau}_2$. Constraints of the form $\beta \sqsupseteq \varphi$ are called *effect* constraints, and mean that the cost of a variable β is above the cost expressed by φ. □

3.3 The Type System

The type system has type variables α, base types for Booleans, integers, and unit, 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 are given meaning by the solution of constraint sets. Types are formed by the following grammar.

$$\hat{\tau} ::= \alpha \mid bool \mid int \mid () \mid task\,\beta\,\hat{\tau} \mid \hat{\tau}_1 \to \hat{\tau}_2$$

3.3.1 Polymorphism, Polyvariance and Subtyping. These three techniques allow different types, and therefore different costs, for the same variable in different contexts. Though not strictly necessary for the system to be sound, they increase its precision by preventing poisoning in the following situations. Polymorphism, as in normal Hindley-Milner systems, allows different occurrences of a variable in the body of a let binding to have different types, provided that the type can be generalized. As types can be chosen to fit the context, so can be the costs.

Polyvariance is a weak form of polymorphism. It is applicable to let-bound variables, and comes into play when the type of a variable can not be generalized. Polyvariance allows the cost of a task to be increased when a context requires it, without affecting the variable's type in other contexts.

Subtyping is useful for lambda-bound variables, which are not subject to polymorphism and polyvariance. It also allows the cost of a task to be increased when a context requires it without affecting the type in other contexts. Subtyping comes into play in branches of conditionals and the arguments of function application.

Standard Hindley-Milner type inference works by solving type equality constraints. Equality constraints can be solved as soon as they are discovered, so most algorithms perform unification on the fly, without ever explicitly generating constraint sets. We work with inequality constraints, which can only be solved once all constraints are known. This is why we have to explicitly construct and keep track of constraints during type inference, and solve them at the end in a separate phase.

The general architecture of constraint generation and solving is still the same as in our previous system, so we just provide a brief summary. The actual constraints and their semantics is different, and is discussed in more detail. The system uses type schemes to implement polymorphism, polyvariance and subtyping. Type schemes σ 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. Type schemes are introduced to the environment Γ by generalization in the [t-let] rule, and eliminated by instantiation ($\succ$) in the [t-var] rule.

The typing rules use the entailment relation ($\Vdash$) for constraints. The entailment relation captures syntactically all relevant semantic aspects of what it means for a constraint set to imply a constraint. The most important rule is set inclusion, that is if $c \in C$ then $C \Vdash c$. The rules are still the same as in our previous system and not covered here.

The subtyping relation ($<:$) is also still the same. Its rules implement full subtyping, with contravariant functions as usual. The most important rule states that subtyping for tasks requires cheaper effects, that is if $C \Vdash \beta_1 \sqsubseteq \beta_2$ and $C \Vdash \hat{\tau}_1 <: \hat{\tau}_2$ then $C \Vdash task\,\beta_1\,\hat{\tau}_1 <: task\,\beta_2\,\hat{\tau}_2$.

3.3.2 Typing Rules. Typing judgements have the form $C; \Gamma \vdash e : \hat{\tau}$ where C is a constraint set, Γ a typing environment, e an expression and $\hat{\tau}$ an annotated type. The typing rules for our annotated type system are given in Figure 11.

The rule scheme **[t-const]** gives the corresponding types to pure constants, that is *bool* to Boolean constants, *int* to integer constants, and the unit type () to the unit value (). **[t-op]** is a rule scheme for all pure operators in the language and typechecks them in the natural way. **[t-var]** looks up the type scheme of a variable in the environment and instantiates it. **[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 reduces to 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-ret]** states that an expression of the form **return** e is a task that can have any cost. **[t-use]** states that basic tasks have the cost as specified in the program. **[t-seq]** states that a sequential composition of two tasks has the cost of the right task appended to the cost of the left one. **[t-par]** states that the parallel composition of two tasks costs as much as the sum of the two tasks individually. **[t-pp]** states how to

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

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

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

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

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

$$\text{[t-app]}\ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : \hat\tau_1 \to \hat\tau_2 \\ C;\Gamma \vdash e_2 : \hat\tau_3 \qquad C \Vdash \hat\tau_3 <: \hat\tau_1 \end{array}}{C;\Gamma \vdash e_1 e_2 : \hat\tau_2}$$

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

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

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

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

$$\text{[t-seq]}\ \frac{\begin{array}{c} C;\Gamma \vdash e_1 : \mathit{task}\,\beta_1\,\hat\tau_1 \\ C;\Gamma \vdash e_2 : \mathit{task}\,\beta_2\,\hat\tau_2 \\ C \Vdash \beta \sqsupseteq \beta_1 +\!\!+\, \beta_2 \end{array}}{C;\Gamma \vdash e_1 \gg e_2 : \mathit{task}\,\beta\,\hat\tau_2}$$

$$\text{[t-par]}\ \frac{\begin{array}{c} C;\Gamma \vdash e_1 : \mathit{task}\,\beta_1\,() \\ C;\Gamma \vdash e_2 : \mathit{task}\,\beta_2\,() \\ C \Vdash \beta \sqsupseteq \beta_1 + \beta_2 \end{array}}{C;\Gamma \vdash e_1\ \&\ e_2 : \mathit{task}\,\beta\,()}$$

$$\text{[t-pp]}\ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : \mathit{task}\,\beta_1\,\hat\tau_1 \\ C;\Gamma \vdash e_2 : \mathit{task}\,\beta_2\,\hat\tau_2 \\ C \Vdash \beta \sqsupseteq (\beta_1 +\!\!+\, \gamma_1) + (\beta_2 +\!\!+\, \gamma_2) \end{array}}{C;\Gamma \vdash \mathit{pp}(e_1\ \&\ e_2, \gamma_1, \gamma_2) : \mathit{task}\,\beta\,\hat\tau_2}$$

Figure 11: The annotated type system

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 process pool is the sum of the combinations of the actual costs so far and the predicted rest.

Example 3.5. This example demonstrates how the system predicts the cost of a simple sequential composition. Consider the following program.

use (1R,1) () $\gg$ **use** (2R,1) ()

Figure 12 shows a typing derivation for this program, where

$$C = \{\, \beta \sqsupseteq \beta_1 +\!\!+\, \beta_2,\ \beta_1 \sqsupseteq \mathrm{sky}(1R,1),\ \beta_2 \sqsupseteq \mathrm{sky}(2R,1) \,\}.$$

A solution of C is shown in Figure 13 (A), where β is an exact prediction of the actual cost of the program.

Note that although this example only has a single skyline, we are implicitly talking about costs, which are families of skylines together with end intervals. This is important, because Figure 13 (B) is also a solution to C, where β_1 is strictly above the required sky$(1R,1)$. In order to satisfy the ordering on costs it must respect the end interval of sky$(1R,1)$, which means it must include 1. The end intervals are drawn under the skylines. □

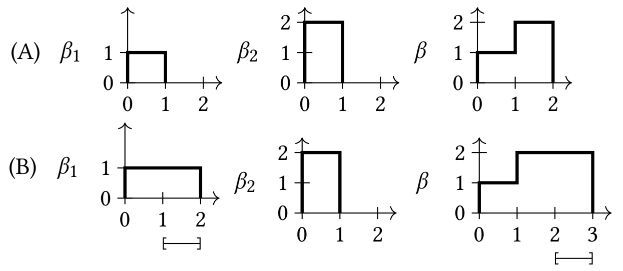


Figure 13: (A) Solution for the constraint set C of Example 3.5. (B) Another solution, overapproximating β_1.

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. 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 core of the algorithm for skylines is largely unchanged compared to our previous system. Most of the new code deals with skylines and operations on them. Unification and subsumption constraint solving are unchanged. Algorithm $\mathcal{W}$ still works the same, save for the places where it generates constraints about skylines. The biggest changes have been made to the effect constraint solver.

4.1 Algorithm W and Unification

Unification takes two types and returns a substitution if the types can be unified, and an error otherwise. Figure 14 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 14. In line **(1)**, tasks are

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

$$\text{[t-seq]}\ \dfrac{\text{[t-use]}\ \dfrac{C;\emptyset \vdash ():() \qquad C \Vdash \beta_1 \sqsupseteq \mathrm{sky}(1R,1)}{C;\emptyset \vdash \mathbf{use}\ [1R,1]\ () : \mathit{task}\,\beta_1\,()} \quad \text{[t-use]}\ \dfrac{C;\emptyset \vdash ():() \qquad C \Vdash \beta_2 \sqsupseteq \mathrm{sky}(2R,1)}{C;\emptyset \vdash \mathbf{use}\ [2R,1]\ () : \mathit{task}\,\beta_2\,()} \qquad C \Vdash \beta \sqsupseteq \beta_1 +\!\!+\, \beta_2}{C;\emptyset \vdash \mathbf{use}\ [1R,1]\ () \gg \mathbf{use}\ [2R,1]\ () : \mathit{task}\,\beta\,()}$$

Figure 12: Typing derivation for a simple sequence.

$$\begin{aligned}
&\mathcal{U}(\mathit{bool},\mathit{bool}) = \mathcal{U}(\mathit{int},\mathit{int}) = \mathcal{U}(\alpha,\alpha) = [\,]\\
&\mathcal{U}(\alpha,\hat\tau) = [\alpha \mapsto \hat\tau] \text{ if } \alpha \notin \mathit{FTV}(\hat\tau), \text{Error otherwise.}\\
&\mathcal{U}(\hat\tau,\alpha) = \mathcal{U}(\alpha,\hat\tau)\\
&\mathcal{U}(\hat\tau_1 \to \hat\tau_2, \hat\tau_3 \to \hat\tau_4) = \theta_2 \circ \theta_1 \text{ where}\\
&\quad \theta_1 = \mathcal{U}(\hat\tau_1,\hat\tau_3)\\
&\quad \theta_2 = \mathcal{U}(\theta_1\hat\tau_2,\theta_1\hat\tau_4)\\
&\mathcal{U}(\mathit{task}\,\beta_1\,\hat\tau_1, \mathit{task}\,\beta_2\,\hat\tau_2) = \theta_2 \circ \theta_1 \text{ where} && (1)\\
&\quad \theta_1 = \mathcal{U}(\beta_1,\beta_2)\\
&\quad \theta_2 = \mathcal{U}(\theta_1\hat\tau_1,\theta_1\hat\tau_2)\\
&\mathcal{U}(\hat\tau_1,\hat\tau_2) = \text{Error: cannot unify types.}\\
&\mathcal{U}(\beta,\beta) = [\,]\\
&\mathcal{U}(\beta_1,\beta_2) = [\beta_1 \mapsto \beta_2] && (2)
\end{aligned}$$

Figure 14: The unification algorithm.

$$\begin{aligned}
&\mathcal{W}(\Gamma,\mathbf{True}) = \mathcal{W}(\Gamma,\mathbf{False}) = \langle \mathit{bool},\ [\,],\ \emptyset\rangle && (3)\\
&\mathcal{W}(\Gamma,n) = \langle \mathit{int},\ [\,],\ \emptyset\rangle && (4)\\
&\mathcal{W}(\Gamma,x) = \langle \hat\tau,\ [\,],\ C\rangle \text{ where } \langle\hat\tau,C\rangle = \mathit{inst}(\Gamma(x)) && (5)\\
&\mathcal{W}(\Gamma,\mathbf{fn}\,x.e_b) = \langle \theta_b\alpha \to \hat\tau_b,\ \theta_b,\ C_b\rangle \text{ where} && (6)\\
&\quad \alpha \text{ fresh}\\
&\quad \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) = && (7)\\
&\quad \langle \theta_{123}\alpha_x \to (\theta_3\circ\theta_2)\hat\tau_r,\ \theta_{123},\ C_2 \cup C_s\rangle\\
&\quad \text{where } \alpha_x,\alpha_r \text{ fresh}\\
&\quad \langle\hat\tau_r,\theta_1,C_1\rangle = \mathcal{W}(\Gamma[f \mapsto \alpha_x \to \alpha_r][x \mapsto \alpha_x],e_b)\\
&\quad \theta_2 = \mathcal{U}(\hat\tau_r,\theta_1\alpha_r)\\
&\quad \langle C_e,C_s,\theta_3\rangle = \mathit{solveSubsumptions}(\theta_2 C_1)\\
&\quad C_2 = \mathit{widen}(\theta_2 C_e)\\
&\quad \theta_{123} = \theta_3\circ\theta_2\circ\theta_1\\
&\mathcal{W}(\Gamma,e_1e_2) = \langle \theta_3\alpha_2,\ \theta_3\circ\theta_2\circ\theta_1, && (8)\\
&\quad (\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}\\
&\quad \alpha_1,\alpha_2 \text{ fresh}\\
&\quad \langle\hat\tau_1,\theta_1,C_1\rangle = \mathcal{W}(\Gamma,e_1)\\
&\quad \langle\hat\tau_2,\theta_2,C_2\rangle = \mathcal{W}(\theta_1\Gamma,e_2)\\
&\quad \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, && (9)\\
&\quad (\theta_4\circ\theta_3\circ\theta_2)C_1 \cup (\theta_4\circ\theta_3)C_2 \cup \theta_4 C_3\\
&\quad \cup \{\, (\theta_4\circ\theta_3)\hat\tau_2 <: \alpha,\ \theta_4\hat\tau_3 <: \alpha \,\}\rangle\\
&\quad \text{where } \alpha \text{ fresh}\\
&\quad \langle\hat\tau_1,\theta_1,C_1\rangle = \mathcal{W}(\Gamma,e_c)\\
&\quad \langle\hat\tau_2,\theta_2,C_2\rangle = \mathcal{W}(\theta_1\Gamma,e_t)\\
&\quad \langle\hat\tau_3,\theta_3,C_3\rangle = \mathcal{W}((\theta_2\circ\theta_1)\Gamma,e_e)\\
&\quad \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_3,\ \theta_3\circ\theta_2\circ\theta_1,\ \theta_3 C_s' \cup C_2\rangle && (10)\\
&\quad \text{where}\\
&\quad \langle\hat\tau_1,\theta_1,C_1\rangle = \mathcal{W}(\Gamma,e_x)\\
&\quad \langle C_e,C_s,\theta_2\rangle = \mathit{solveSubsumptions}(C_1)\\
&\quad \Gamma' = (\theta_2\circ\theta_1)\Gamma\\
&\quad \langle\sigma_1,C_s'\rangle = \mathit{generalize}(\hat\tau_1,\Gamma',C_e \cup C_s)\\
&\quad \langle\hat\tau_3,\theta_3,C_2\rangle = \mathcal{W}(\Gamma'[x \mapsto \sigma_1],e_b)
\end{aligned}$$

Figure 15: The general purpose part of algorithm $\mathcal{W}$.

unified by unifying their annotation variables and their return types. In line **(2)**, annotation variables are unified by generating a substitution.

Algorithm $\mathcal{W}$ 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 actual cost of e. Algorithm $\mathcal{W}$ is given in Figures 15 and 16. Figure 15 shows type inference for pure expressions. Line **(3)** and **(4)** typecheck Boolean and integer constants. Such expressions are of type *bool* and *int* respectively.

The clause for variables in line **(5)** 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 **(6)** puts the bound variable x into the environment with a fresh type variable α and typechecks the function body.

Recursive functions, line **(7)**, are typechecked by putting the function and the argument with fresh type variables into the environment and then checking the function body. Then, subsumption constraints are solved. This extracts as much information as at that point possible out of the constraint set. The result of subsumption constraint solving is a set of effect constraints C_e, a set of unresolved subsumption constraints C_s, and a substitution resulting from unification. The resulting effect 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

$$\begin{aligned}
&\mathcal{W}(\Gamma, \textbf{use}\ [k,d]\ e) = &(11)\\
&\quad \langle task\,\beta\ \hat{\tau}_1, \theta_1, \{\ \beta \sqsupseteq sky(k,d)\ \} \cup C_1 \rangle\\
&\quad \text{where } \alpha, \beta \text{ fresh}\\
&\quad \langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e)\\
&\mathcal{W}(\Gamma, e_1 \gg\!= e_2) = \langle task\,\beta\ ((\theta_4 \circ \theta_3)\alpha_2), &(12)\\
&\quad \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\\
&\quad (\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2\\
&\quad \cup \{\ \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) +\!\!+ (\theta_4\beta_2)\ \}\rangle\\
&\quad \text{where}\\
&\quad \langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)\\
&\quad \langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)\\
&\quad \beta, \beta_1, \beta_2, \alpha_1, \alpha_2 \text{ fresh}\\
&\quad \theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1,\ task\,\beta_1\ \alpha_1)\\
&\quad \theta_4 = \mathcal{U}(\theta_3\hat{\tau}_2,\ (\theta_3\alpha_1) \to task\,\beta_2\ \alpha_2)\\
&\mathcal{W}(\Gamma, e_1\ \&\ e_2) = \langle task\,\beta\ (), &(13)\\
&\quad \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\\
&\quad (\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2\\
&\quad \cup \{\ \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) + (\theta_4\beta_2)\ \}\rangle\\
&\quad \text{where}\\
&\quad \langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)\\
&\quad \langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)\\
&\quad \beta, \beta_1, \beta_2 \text{ fresh}\\
&\quad \theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1, task\,\beta_1\ ())\\
&\quad \theta_4 = \mathcal{U}(\theta_3\hat{\tau}_2, task\,\beta_2\ ())
\end{aligned}$$

Figure 16: The domain specific part of algorithm $\mathcal{W}$.

constraint is widened with its own right hand side. Subsumption constraints are left untouched. Formally:

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

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 5.6.

The clause for applications, line **(8)**, 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 **(9)**, 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 **(10)** first typechecks the defining expression of x, solves subsumptions as far as possible, 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 rest of the algorithm is shown in Figure 16.

The clause in line **(11)** 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 **(12)**, 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 append of the cost of the left and right arguments.

Parallel task composition, line **(13)**, is simpler than sequential composition. Both arguments must be tasks returning unit. The value of the overall expression is a task returning unit. The cost of the overall expression is the sum of the costs of the arguments.

4.2 Subsumption Constraint Solving

The subsumption constraint solver is still the same as in our previous system, and not covered in detail here. Subsumption constraint solving happens during and after algorithm $\mathcal{W}$, and its goal is to decompose subsumption constraints as far as possible to extract effect constraints according to the subtyping rules. The subsumption constraint solver takes as input the constraints that algorithm $\mathcal{W}$ collects and returns a triple $\langle C_e, C_s, \theta \rangle$. C_e is a set of effect constraints that have been identified by solving subtyping constraints. C_s is a set of unresolved subsumption constraints. These are only relevant for local solving in the rules for recursive functions and let expressions. The substitution θ is the result of all unifications that had to be performed during solving.

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], with one modification. All our skylines start at time 0 in their local coordinate system, and appending them to other skylines shifts them in time. Whether the information content of a single constraint has been fully incorporated into the final solution can no longer be determined by looking at that constraint in isolation. It is therefore not possible to determine whether a constraint has to be evaluated again in the worklist algorithm.

To solve this problem we make use of the fact that in a complete lattice, these two formulations are equivalent: $x \sqsupseteq y \land x \sqsupseteq z \iff x \sqsupseteq y \sqcup z$.

Definition 4.2. (Combining constraints) Let C be a constraint set with possibly multiple constraints for each annotation variable β. The *combined constraints* C' have a single constraint for every β, such that if

$$\begin{aligned} \beta \sqsupseteq \varphi_1,\ \ldots,\ \beta \sqsupseteq \varphi_n \in C\\ \text{then } \beta \sqsupseteq \varphi_1 \sqcup \ldots \sqcup \varphi_n \in C'. \end{aligned}$$

All constraints for every β are combined into a single constraint. □

Our worklist algorithm takes as input the combined constraints of the constraints that algorithm $\mathcal{W}$ computes, and returns a solution of them. The algorithm works as follows.

Initially, all constraints are marked as dirty, so they are evaluated at least once. The worklist iteration then starts evaluating dirty constraints one-by-one. After it evaluates a constraint for a variable β, it compares the value to the value of the previous round. If these two differ, the algorithm has learned something new about β, and marks all constraints that depend on β as dirty. When there are no more dirty constraints, the algorithm terminates.

In every evaluation of a constraint for a variable β, the algorithm learns everything that is known so-far about β. This allows the algorithm to decide that it has learned something new about β if the value of the constraint is *different*, even if it is not strictly *above* the previous value. This is important because as solving progresses, cost spikes tend to wander to the right to their final place. As they move further to the right, they disappear where they have been in the previous iteration. The new iteration is not strictly above the old iteration, yet the algorithm has learned something new.

Widening. Widening is the final piece of the puzzle, and the reason why the worklist algorithm always eventually terminates. Widening is a heuristic used by the effect constraint solver to compute solutions with infinity. The inputs to widening are two consecutive iterations of the same constraint. They correspond roughly to loop unrollings. The output of widening is a skyline that is idempotent under further iterations, representing an upper bound to the cost of the loop.

Widening needs to discard information to guarantee termination of the worklist iteration, while retaining enough information for the results to have some value. Widening applies to costs, and as such it must widen skylines and end intervals.

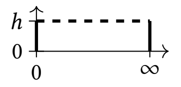
Definition 4.3. (Widening of end intervals)

$$[x_1, y_1] \nabla [x_2, y_2] = \begin{cases} [x_1, \infty] & \text{if } y_2 > y_1 \\ [x_1, y_1] & \text{otherwise} \end{cases}$$

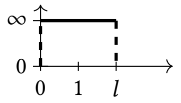
The beginning of the end interval of a loop can never decrease, so the worst case is always the beginning of the left argument. If the end of the end interval grows, widening jumps to infinity. □

Definition 4.4. (Widening of reusable skylines) Let s_1, s_2 be two iterations of a reusable skyline s.

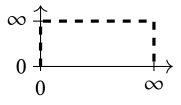
$s_1 \nabla s_2 =$



if s grows in length but not in height



if s grows in height but not in length



if s grows in length and height

s_1 otherwise

Where h and l are the maximum height and length of s_1. We say s grows in length if s_2 is longer than s_1. Similarly s grows in height if the maximum height of s_2 is greater than the maximum height of s_1. □

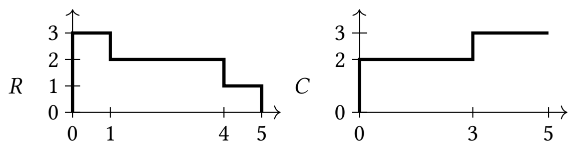
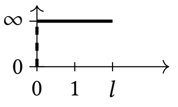


Figure 17: Costs for example 5.1

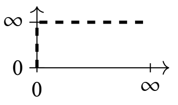
Definition 4.5. (Widening of consumable skylines) Let s_1, s_2 be two iterations of a consumable skyline s.

$s_1 \nabla s_2 =$

impossible if s grows in length but not in height



if s grows in height but not in length



if s grows in length and height

s_1 otherwise □

5 DISCUSSION

In this section we discuss the analysis results of some example programs.

Example 5.1. (Exact predictions) When the control flow of a program does not depend on computations, the predicted cost matches the actual cost exactly, as in the following program.

```
let t1 = use [1R,1] 0 in
let t2 = use [1R+1C,4] 0 in
let t3 = use [1R+1C,3] 0 in
let t4 = use [1R+1C,2] 0 in
(t1 & t2) & (t3 ≫ t4)
```

This program is a mix of parallel and sequential compositions. The cost is shown in Figure 17.

Example 5.2. (Overapproximation by conditional) The following program demonstrates how the analysis overapproximates costs in both height and length.

```
let t1 = use [1R,1] 0 in
let t2 = use [3R,2] 0 in
let t3 = use [2R,1] 0 in
(if True then t1 else t2) ≫ t3
```

The overapproximation is caused by a conditional. There is a large cost in the else branch which is not executed, but the analysis must take it into account. Task *t3* can start either at time 1 or at time 2, depending on which branch of the conditional is taken. Again, the analysis must anticipate both possibilities, and it does so by using end intervals. The actual cost is *t1* followed by *t3*, which is cheaper and does not take as long as the worst case. Predicted and actual costs are shown in Figure 18.

Example 5.3. (Overapproximation by poisoning) Poisoning happens when a variable is used in different contexts, which all influence its cost. There are several techniques in the literature to

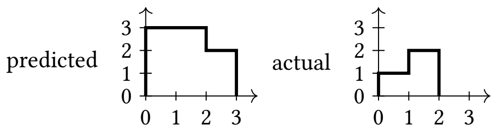


Figure 18: Costs for example 5.2

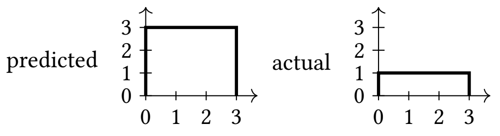


Figure 19: Costs for example 5.3

reduce poisoning, and we have included a couple of them, namely polymorphism, polyvariance, and subtyping. One source of poisoning that still exists are lambda-bound functions. Consider the following program.

```
let const = fn x . fn y . x in
(fn id . const (id (use [1R,3] 0)) (id (use [3R,3] 0))
) (fn x . x)
```

The identity function is lambda-bound and used in two different contexts. It is applied to an expensive task and a cheap task. The application to the expensive task is ignored by the function *const*, but it still influences the type of *id*. The type of *id* is not subject to polymorphism, because it is lambda-bound, and polymorphism only applies to let-bound variables. It is also not subject to subtyping, because subtyping only applies to the argument expression in function applications, not the function expression. The type of the identity function here is from expensive task to expensive task, so sending the cheap task through the identity function makes it expensive. Predicted and actual costs of this program are shown in Figure 19.

Example 5.4. (Widening in length) Another source of overapproximation are recursive functions. Consider the following program. The program uses *loop* to execute task *t1* three times, and then executes task *t2*.

```
let t1 = use [1R,1] 0 in
let t2 = use [2R,1] 0 in
let loop = fix f x . if (x ≤ 1)
  then t1
  else t1 ≫ f (x−1) in
loop 3 ≫ t2
```

Our analysis cannot know how many times the loop will be executed, and has to approximate it with infinity. The analysis does see that *t1* is executed at least once. The predicted cost of the expression "loop 3" is therefore an infinite skyline of height 1, with end interval $[1, \infty]$. The end interval says that *t2* can start as early as time 1, but possibly also at any later time. The predicted and actual costs are shown in Figure 20.

Example 5.5. (Widening in height) The following program executes the task *t1* a number of times in parallel.

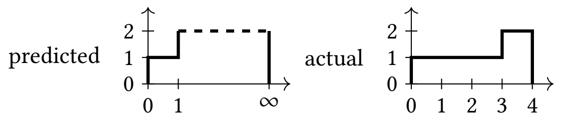


Figure 20: Costs for example 5.4

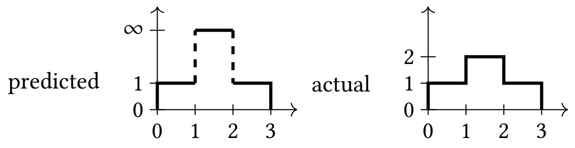


Figure 21: Costs for example 5.5

```
let t1 = use [1R,1] 0 in
let parallel = fix f x . if x ≤ 1
  then t1
  else t1 & f (x−1) in
t1 ≫ parallel 2 ≫ t1
```

Our analysis cannot know how many times *t1* will be executed in parallel, so it has to overapproximate the cost with infinity. It does know that the duration of the parallel will always be 1. The predicted and actual costs are shown in Figure 21.

Example 5.6. (Limitations of the algorithm) There are programs that our implementation cannot handle. For different reasons, the following two programs generate recursive constraints that are missed by widening, therefore causing the effect constraint solver to diverge. The first is a limitation of the type system, the second a limitation of the implementation.

In the following program, recursive constraints are the result of a non-recursive function.

```
let iterate2 = fn f . fn x . f (f x) in
let twice = fn t . t ≫ t in
iterate2 twice (use [1R,1] 0)
```

The function *iterate2* iterates its argument f two times on x. In the application "f (f x)", algorithm $\mathcal{W}$ unifies the argument and result types of f. This results in recursive constraints. The problem here is not that these recursive constraints escape widening, but that they are created in the first place. The two occurrences of f should have different types, which is not possible because f is lambda-bound. A solution to this problem would be higher-ranked polymorphism, which comes with its own problems.

Another source of recursive constraints are polymorphic recursive functions that only later get instantiated to task types. Consider the following program.

```
let iterate = fix f n . fn g . fn x . if n ≤ 0
    then x
    else f (n−1) g (g x) in
let twice = fn t . t ≫ t in
iterate 2 twice (use [1R,1] 0)
```

The function *iterate* iterates its argument g n-times on x. When algorithm $\mathcal{W}$ typechecks *iterate*, which is polymorphic, it does not see any effect constraints, and therefore cannot apply widening. Widening as we have implemented it can only be applied to effect constraints. The subtyping constraints of *iterate* are recursive, and when they get instantiated to effect constraints in the call "iterate 2 twice", we end up with un-widened recursive effect constraints.

A solution to this problem would be to mark subtyping constraints as belonging to recursive functions, so that widening can be applied to them when they become effect constraints.

6 CONCLUSIONS

We have designed and implemented an annotated type system that can predict costs over time of a small iTasks-like workflow calculus. We do not have a correctness proof yet, and given the subtleties of operations on costs we imagine that it is a substantial amount of work. We do have an extensive test suite for the implementation, which includes many hand-crafted examples alongside automatically generated tests using Gast [Koopman et al. 2002]. The correctness of the implementation is tested by running the programs in an interpreter and checking that it never exceeds the prediction.

7 FUTURE WORK

There are a couple of ways to extend the system. One idea is to integrate the analysis into iTasks. At the moment, the analysis is defined on a condensed version of Clean and iTasks, but we would like to make the analysis available to iTasks programmers. This involves hooking into the compiler and supporting language features like data types, pattern matching, and mutual recursion. This line of work also includes integration with Tonic. Tonic [Stutterheim et al. 2014] is a system that can visualize iTasks programs graphically and allows inspecting their progress at run time. In particular we would like to display skylines alongside Tonic diagrams to show predicted costs next to control structure.

Another idea would be to trace points in a skyline back to the places in a program where they may come from. This would be particularly interesting in combination with Tonic. A user could interactively click on a point in a skyline, and the associated tasks in the Tonic diagram are highlighted.

8 RELATED WORK

Our work follows the line of work by Nielson and Nielson, most notably their textbook on program analysis [Nielson et al. 1999], and their papers on polymorphic subtyping [Amtoft et al. 1997; Nielson et al. 1996a,b]. We drew a lot of inspiration for the operational semantics from the paper about the IO monad in Haskell [Peyton-Jones 2001]. There are many papers on program analysis using type and effect systems. The ones that influenced our work are Gedell et al. [2006], the paper about 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 in the field of automated planning [Ghallab et al. 2004]. Works on resource consumption of programs often focus on computational resources, like complexity or memory usage. Two noteworthy papers 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 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. [2012].

ACKNOWLEDGMENTS

We would like to thank Jurriaan Hage, Tim Steenvoorden, Jurriën Stutterheim, Kelley van Evert, Terry Stroup, Bas Lijnse, and Ynske Aerts 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. 1997. Polymorphic Subtyping for Effect Analysis: The Dynamic Semantics. *LNCS* 1192 (1997).

Tobias Gedell, Jörgen Gustavsson, and Josef Svenningsson. 2006. Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis. In *APLAS 2006, Sydney, Australia*. Springer, 200–216.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. 2004. *Automated planning - theory and practice.* Elsevier. I–XXVIII, 1–635 pages.

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. 2007. A generic usage analysis with subeffect qualifiers. In *Proceedings of ICFP 2007, Freiburg, Germany, October 1-3, 2007.* ACM, 235–246.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. *ACM SIGPLAN Notices* 45, 1 (Jan. 2010), 223–236.

Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. 2014. A Hoare Logic for Energy Consumption Analysis. In *Proceedings of FOPARA'13*, Vol. 8552. Springer, 93–109.

Markus Klinik, Jurriaan Hage, Jan Martin Jansen, and Rinus Plasmeijer. 2017. Predicting resource consumption of higher-order workflows. In *Proceedings of PEPM 2017, Paris, France, January 18-20, 2017.* ACM, 99–110.

Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. 2002. Gast: Generic Automated Software Testing. In *IFL (Lecture Notes in Computer Science)*, Ricardo Pena and Thomas Arts (Eds.), Vol. 2670. Springer, 84–100.

Ruud Koot and Jurriaan Hage. 2015. Type-based Exception Analysis for Non-strict Higher-order Functional Languages with Imprecise Exception Semantics. In *Proceedings of PEPM, Mumbai, India, January 15-17, 2015*, Kenichi Asai and Kostis Sagonas (Eds.). ACM, 127–138.

Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. 2012. Incidone: A Task-Oriented Incident Coordination Tool. In *Proceedings of ISCRAM*.

Flemming Nielson, Hanne Riis Nielson, and Torben Amtoft. 1996a. Polymorphic Subtyping for Effect Analysis: The Algorithm. In *LOMAPS (Lecture Notes in Computer Science)*, Mads Dam (Ed.), Vol. 1192. Springer, 207–243.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer.

Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. 1996b. Polymorphic Subtyping for Effect Analysis: The Static Semantics. In *LOMAPS (Lecture Notes in Computer Science)*, Mads Dam (Ed.), Vol. 1192. Springer, 141–171.

Simon Peyton-Jones. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction.* IOS Press, 47–96. Marktoberdorf Summer School 2000.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *PPDP*, Danny De Schreye, Gerda Janssens, and Andy King (Eds.). ACM, 195–206.

Rinus Plasmeijer and Marko van Eekelen. 2002. Clean language report (version 2.1). (2002). http://clean.cs.ru.nl.

Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In *Trends in Functional Programming (Lecture Notes in Computer Science)*, Vol. 8843. Springer, 122–141.

Pedro B. Vasconcelos and Kevin Hammond. 2003. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, Vol. 3145. Springer, 86–101.

Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. 2014. Security type error diagnosis for higher-order, polymorphic languages. *Sci. Comput. Program* 95 (2014), 200–218.