A Formal Type-Centric Framework for Verification and Resource Allocation in Pervasive Sense-and-Respond Systems[†]

Michael Ocean
Endicott College[*]
Computer Science
Beverly, MA
mocean@endicott.edu

Assaf Kfoury
Boston University
Computer Science
Boston, MA
kfoury@cs.bu.edu

Azer Bestavros
Boston University
Computer Science
Boston, MA
best@cs.bu.edu

Abstract

*A shared Sense-and-Respond infrastructure that is embedded into a physical environment requires considerable run-time support to facilitate the dynamic dispatch and execution of new service instances. Such an infrastructure must also be able to statically analyze new services in order to verify their safety and derive their specific resource requirements (*i.e.*, prior to dispatch). Toward this goal we have developed a multi-dimensional type system for our pervasive sensory service composition language; this formalism extracts implicit constraints from service instances to verify an expanded notion of type safety. While our formal system is rigorous, it is light-weight and essentially transparent to a service programmer. The type-system automatically infers data types that are annotated with a vector of type specific attributes and uses these annotations to establish and verify a range of resource constraints (bounds for computation and memory usage, camera resolution requirements, etc.). In this paper we present an overview of our formal methodology, provide concrete examples of how these formalisms are used in practice (through service logic examples and derived constraint sets) and discuss the details of our implementation.*

1. Introduction

The Sensor Network WorkBench (SNBENCH [7]) is a collection of compile-time tools and run-time components that enable the painless development and deployment of Sense-and-Response services that run on a shared infrastructure. Toward SNBENCH's goal of enabling unsophisticated users to compose these services we provide our users with high-level languages that are compiled down to a functional-style Domain Specific Language (DSL), called STEP (Sensor Task Execution Plan). STEP is resource agnostic insofar as service logic may refer to particular types of resources (*e.g.*, an image sensor) without indicating which *specific* resources should be utilized within the service.

Our ability to allocate resources on which to deploy STEP services is contingent upon our ability to verify the safety of new services and to derive resource requirements from new service instances. While the motivation of the SNBENCH project and specific implementation details for its vision, architecture and use have been published elsewhere ([4],[16] and [15] respectively), in this paper we present the static analysis techniques that we have developed to provide safety and resource constraint extraction on our sensing-centric language, STEP.[1] STEP is analogous to a common Instruction Set Architecture; by providing analysis techniques at the level of STEP, we extend this functionality to any and all higher-level languages.

We base our type system on sized (*a.k.a.* static-dependent) type systems, wherein upper bound size annotations on types coupled with cost functions are used to determine memory (storage) and processing (worst case execution) bounds. We expand our size tracking to multiple dimensions (*i.e.*, multiple dimensions of size annotations) toward the goals of (1) supporting images as a first-class data type and (2) enabling the static inference of required image (and image sensor) resolutions from implicit constraints.

Unlike traditional scalar data, both size bounds of an image (*i.e.*, upper and lower, where the lower bound is the potential minimum image resolution) may have an impact on functional correctness. For example, attempting to rec-

[†] This work was supported partially by NSF Awards #0820138, #0720604, #0735974, and #0524477. Any opinions, findings, conclusions, or recommendations expressed in this work do not necessarily reflect the views of the NSF.

[*] This author's work was conducted primarily while at Boston University.

[1] In this paper we present only a subset of our formal system and omit the proof of Soundness (Progress and Preservation); the complete formalism appears in a Technical Report [14].

978-1-4244-3721-4/09/$25.00

ognize a face in a low-resolution image may never succeed, or worse, might diverge depending on the implementation. While one could consider adding additional types and subtyping relations to the type system to support awareness of image resolutions (*e.g.*, LowResolutionImage, MediumResolutionImage, HighResolutionImage) it should be obvious that this sort of solution does not scale.

Our sized type system can bound costs for memory and computation, and also produces sensing domain specific resource constraints. In tracking bound both upper size bounds and lower size bounds we are able to make statements that bound a worst-case execution time and also provide bounds for image resolution; the latter property ultimately leads to establishing the correctness of image processing expressions.

A Motivating Example

Our goal is to be able to leverage the size annotations in the type system to provide both an upper-bound for computational requirements of services (as prior works have done), while additionally (1) maintaining minimum size aspects to verify correctness in the presence of functions that require a minimum size to ensure correctness and (2) determining and maintaining implicit constraints on resources and data sizes as extracted from contextual usage in a given service instance.

For example consider the code fragment below:

```
(* Every 100 milliseconds, try to detect motion
   in an image taken from a random camera.
   If motion is detected, send an e-mail. *)

 letonce IMG = get(sensor(IMAGE,ANY)) in
  period(100ms,
    trigger(facedetect(IMG),email("mocean",IMG))
  )
```

In the code given, the variable `IMG` represents an image captured from "any" image sensor. However not all image sensors (*i.e.*, cameras) have the same capabilities with respect to image resolution (*e.g.*, a webcam might capture images at a resolution several times lower than that of an embedded Pan-Tilt-Zoom camera). In this program instance there are *implicit* constraints on the image that indicate that not just *any* sensor will do. The function (or as we call it, Opcode) **facedetect** constrains the size of `IMG`, as it requires a minimum resolution to correctly detect a face in an image. A **trigger** construct specifies repetitive conditional evaluation. It repeatedly evaluates a boolean expression (until it is true) and then evaluates a second expression (the first expression "triggers" the second). The explicit periodicity function (or as well call it, Flowtype) **period** indicates that this expression must run every 100 milliseconds. Thus there are two constraints on the resolution of the acquired image; the resolution must satisfy the minimum requirements imposed by the face detection operation, yet also be small enough to allow computation to occur every 100 milliseconds. These constraints on the resolution of the image must propagate back to the image sensor from which the image will be acquired to ensure that the sensor reserved for this program can support the required resolution (or range of resolutions).

Our sized constraint set (when solved) can be used to (1) guide task assignment (*e.g.*, do not split computation over the network where the data size will incur a steep networking overhead), (2) guide resource allocation (*e.g.*, reserve the correct sensor determined from a resolution range derived from use in context), and (3) determine if a program is fundamentally or temporarily infeasible (*e.g.*, some specific resolution too low to perform computations, required periodicity cannot be met given current available resources). While it may not be clear that this particular, simple example requires a strict periodic deadline, our environment also targets tasks that provide coordination across various time-sensitive entities (*e.g.*, video sensors that guide robots).

As far as we are aware, we are the first project to employ such static verification techniques in this domain. The static methods presented here have benefit beyond their application to image data, and may be extended to other data types that have multiple aspects (e.g., image quality, video frame-rate or aspect ratio, audio sampling rate).

2. The SNBENCH Paradigm

To orient the reader to the platform to ease further discussion, in this section we briefly highlight the salient features of SNBENCH. The vision, goals and high-level overview of the SNBENCH infrastructure have been reported elsewhere [4]. Implementation details may be found in [16] and a study of its successful use in the domain of combined physical and network security are available in [15].

Overview: SNBENCH consists of programming support and a runtime infrastructure for Sensor Networks comprised of heterogeneous sensing and computing elements that are physically embedded into a shared environment. We refer to such a physical space with an embedded SN as a Sensorium [6]. The SNBENCH framework allows Sensorium users to easily program, deploy, and monitor the services that run in this space while insulating the user from the complexity of the physical resources therein. The support that SNBENCH extends to a Sensor Network is akin to the support that higher-level languages and operating systems provide to traditional, single machine environments (language safety, APIs, virtualization of resources, scheduling, resource management, *etc*). SNBENCH is extensible by design such that new hardware and software capabilities may be painlessly folded into the infrastructure by its advanced users and those new capabilities easily leveraged by its novice users.

Programming Life Cycle: SNBENCH provides a high-level programming language with which to specify programs (services) that are submitted to the resource management component which in turn disseminates program fragments to the run-time infrastructure for execution. At the lowest level, each sensing and/or computing element hosts a Sensor eXecution Environment (SXE) that abstracts away specific details of the host and attached sensory hardware. SXEs are assigned tasks by the resource management components of SNBENCH; the Sensorium Service Dispatcher and Sensorium Resource Manager in tandem monitor SN resources, schedule (link) and assign (bind) tasks to available SXEs. A graphical representation of this end-to-end support is shown in Figure 1.

The Virtual Instruction Set Architecture of SNBENCH is the Sensorium Task Execution Plan (STEP), a domain specific tasking-language used to describe complete programs and fragments alike. A STEP program is a graph of a SN programs's data-flow and computational dependencies, where the nodes of a STEP graph represent the atomic computation and sensing operations and edges represent data flow. In execution, demand for evaluation pushes down from the root of the graph to the leaves and values percolate up from the leaves back to the root. STEP nodes describe data, control flow (*e.g.*, repetition, branching) and computation operations that we refer to as STEP Opcodes and the SXE maintains implementations of the Opcodes with which it may be tasked.

Run-Time Support: STEP tasks (or subtasks) are assigned (linked) to the available SXEs for execution by the Service Dispatcher. The Service Dispatcher must, therefore, be able to characterize the STEP tasks to ensure that they are valid, and to estimate their resources requirements for the purpose of allocation. For illustration's sake, we make a distinction between uncertified STEP tasks and those that have been certified (*i.e.*, validated by our type system as correct and annotated with resource requirements). Where certification occurs is ultimately immaterial provided that the information is available and can be trusted (*i.e.*, cannot be forged).

Verification Support: To allow the physical sensing, computation, and actuation resources to be simultaneously shared by services provided by the occupants of the space (who are expected to lack formal training in software development), it is critical that new services are vetted prior to dispatch (i.e., statically) to bound their potential resource consumption and verify safety. In the SNBENCH paradigm, novice users program their tasks in accessible, higher-level languages that are compiled into a functionally equivalent representation in STEP. Providing verification to the STEP language, the common instruction set of our language hierarchy, ensures that all higher level languages may enjoy the benefits of this analysis. It is a challenge to effectively report meaningful warnings and/or error messages to the user when analyzing a target (compiled) language rather than the source. To help close this gap our initial analysis implementation records the line numbers in the source language with each constraint such that, at a minimum, we may point the user to the correct location in the event that a problem is detected.

3 Related Work

Finding upper bounds on the execution time of programs and program fragments is a well-established problem in computing [20]. For example, the static analysis provided by the aiT tool (described in [19], and elsewhere), provides worst and best case execution time bounds using control-flow analysis and abstract interpretation. We narrow our consideration of related works to those that also use type-systems for resource constraint discovery and assessment. We distinguish ourselves from many of these works, in some part, in that we statically infer and inject viable specifications into vaguely specified programs (*i.e.*, resource agnostic code).

Our work has been largely inspired by existing works that estimate an upper bound on execution time and memory requirements via a formal type system that has been annotated with upper size bounds on data types. These works can be viewed as belonging to what have been called Dependent Type Systems.

We have made the conscious decision to apply static verification to our single target tasking language which may be in turn translated into other representations for execution on the target platform (*e.g.*, the native language of the target device). This approach is contrasted with works in other domains that provide verification to the lowest level language. Our approach provides verification benefits independent of the source language or the target language (or the target platform, for that matter). Examples of applying static verification at the lowest (*i.e.*, Assembly language) level include the Typed Assembly Language [13] and a closer comparison can be made to the Dependently Typed Assembly Language [22] or its current incarnation as ATS (Applied Type System) [21]. Ultimately these works, insofar as they are focused on the lowest level language, are naturally used to verify properties that are relatively outside the scope of the Sense-and-Respond environments that we target (*e.g.*, the computation of array size bounds to prevent memory leaks is different from determining image sensor constraints for resource allocation).

Works that define Dependent Type Systems for bounding execution time on higher-level functional languages include (but are not limited to) Static Dependent Costs [17], Sized Type Systems [10], and Sized Time Systems [12]. Indeed, there has been a large interest in applying custom type systems to domain specific languages (which peaked in the late

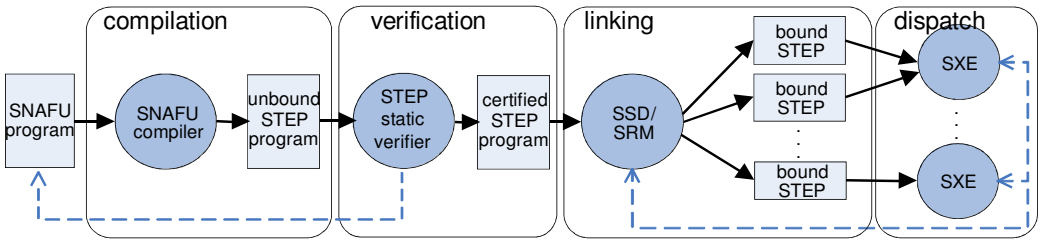

Figure 1. The SN program life-cycle as enabled by snBench. Rectangles represent data, circles represent tasks/processes, and the dashed lines represent control communication (dependency).

nineties, *e.g.*, the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999). Later type systems have been used to bound other resources such as expected heap space usage (*e.g.*, [9], [3]).

While many Dependent Type Systems directly target resource bounding for the real-time embedded community (*e.g.*, the current incarnation of the Sized Time System [8], Mobile Resource Guarantees for Smart Devices [3]), SNBENCH's support for distributed image processing requires the construction of new formal methods. Additionally the operating environment of SNBENCH is intrinsically distributed and time dependent and, as such, the techniques of Dependent Typing require more than small adjustments to facilitate the verification of type safety and the extraction of execution time bounds.

Our language (and infrastructure) supports the direct modification of image data which, unlike traditional scalar data, has an overloaded notion of size (*i.e.*, resolution) that has a direct impact on functional correctness. In this environment the type signature of a function must include explicit resolution (size) bounds to convey what size *ranges* of data a function can *correctly* process. In prior works size annotation has nothing to do with functional correctness; we recognize a need to track a lower size bound (annotation) in addition to the upper size bound, and from this need, we have established a system in which the size annotations are *multi-dimensional*. While the formalism described in Section 4 only includes a lower and upper size bound, Section 7 discusses how easily more dimensions could be added and gives examples.

Additionally our system uses size constraints to solve for data size when it is not explicitly specified by the programmer (*i.e.*, size annotation variables). Our constraint set is explicit within the typing rules yet constraints are derived implicitly from program code, using our constrained size type signature for primitive operators. In solving the constraint set we can deduce feasible (and/or optimal) data sizes which directly map to image resolutions, resource constraints and sensor capabilities for image manipulation programs. Finally, we allow primitive operators that directly manipulate the constraint set, allowing the programmer to explicitly constrain types without influencing the execution (so called, Flowtypes). We are unaware of any other work that treats images as first class datatypes or that uses a type system to statically create such size constraint relationships to deduce required image sizes and sensor capabilities.

4. The Formal Framework

In this section we highlight the salient aspects of the formal logic that underlies the verification component described above. Due to space constraints, we present only a subset of the complete formalism and omit the proof of Soundness (= Progress Theorem + Preservation Theorem). These complete details are available in a Technical Report [14].

4.1 Typing and Sizing of Expressions

The typing (static semantic) of an expression takes the general form:

$$\Gamma \vdash e : t^{\{s_{min}, s_{max}\}} \$c, \kappa$$

Where e is an expression, t is a base type (*e.g.*, *Int*, *Bool*, *Img*), the pair $\{s_{min}, s_{max}\}$ is the *size* annotation for the type (s_{min} is the lower size bound, s_{max} is the upper size bound), c is an approximation of the computational cost of the expression and κ is a size constraint set (s_{max} and s_{min} are size annotations constrained by the simple equations stored in κ, as we will see later). In English we read this as: "Expression e has a worst-case computational cost of c and is of type t under the typing environment Γ, where t has a minimum size s_{min}, a maximum size s_{max}, and is subject to size constraints κ." The size annotation, described here as a pair, is a tuple that may be expanded to track any number of dimensions/aspects (as described in Section 7).

Over the course of the typing derivation of a complex expression, annotations will accumulate constraints in the constraint set κ. The goal of our type system is to build and

solve (*i.e.*, unify) the constraint set following type inference. The solution of this constraint set will either provide bounds for those expressions whose sizing annotations have been omitted (*i.e.*, are variables), or reflect that the constraint set is infeasible for the given program.

For some insight as to how these terms are used in the system, consider expressions that are constants (fixed values). The computational cost (c) for a fixed value is always 0. A constant integer value has both size annotation variables constrained to the integer's actual value.

For image data, the general form of the annotation pair specifies the range of possible resolutions for the image (the number of pixels the image may contain). Resolution in this context does not include the aspect ratio or color-depth, which could be modeled as additional annotations in the tuple (and would be nil for non-image data). As with integers, both values will be the same for a specific instantiation of an image; the typing rule for constant images (*i.e.*, a single image frame) is presented below.

$$\text{(T-IMG)}\ \frac{}{\Gamma \vdash \mathtt{i} : \mathit{Img}^{\{r,r\}}\ \$\ 0, \{r = \mathtt{resolution(i)}\}}$$

A Sensor is a container type; a sensor of type *Sensor* τ will return values of type τ when read. The Sensor type has a negligible, and therefore omitted, static size however the inner (contained) type is annotated with size bounds to indicate the capabilities of the sensor (*e.g.*, the range of resolutions a camera can support). The typing rule for sensors given below reflects that an image sensor that can return images ranging in size from $\mathtt{r}_{min}$ to $\mathtt{r}_{max}$, where these values correlate with the minimum and maximum resolution capabilities of physical hardware. This judgment has no premise and introduces the size variables r_1 and r_2 into the derivation tree; once derivation is complete, unification will solve the constraint set to find a range of valid values for these variables in particular.

$$\text{(T-IMGSENSOR)}\frac{}{\Gamma \vdash \mathtt{image} : \mathit{Sensor\ Img}^{\{r_1,r_2\}}\ \$\ 0, \kappa_r}$$

$$\text{where } \kappa_r = \{r_1 \geq \mathtt{r}_{min}\} \cup \{x_2 \leq \mathtt{r}_{max}\}$$

4.2 Subtyping relations and Images

We define a multi-dimensional subtyping relation that, in one dimension, is similar to the subtyping relation ($\trianglelefteq$) in [12] which allows a weakening of the type to increase a size bound in order to provide an upper bound of estimation of work to be completed. In our environment, however, we notice that the correctness of image processing functions may be impacted by the size of the input (*i.e.*, resolution of the image); as such we cannot arbitrarily increase the logical size of data (which in the case of images is tantamount to increasing the resolution of an image) without adverse consequences to functional correctness.

There is a tension between the desire to provide an upper work bound (which would suggest that the subtype relation for images would enable arbitrary increases to the size of an image) and the desire for the type system to suggest viable image resolutions when resolution is omitted (based on constraints that image manipulation operations place on their input size). Thus, our type annotations are multi-dimensional and, in this example, we track both the upper and lower bounds of sizes. The need to track the lower bound extends into all aspects of the type system, (*i.e.*, annotating all expression types, not just images) and so we use a general sizing/weaken rule (S-SIZED) to describe the subtype relationship for these specific size annotations.

$$\text{(S-SIZED)}\ \frac{(s_{min} \geq s'_{min}) \quad (s_{max} \leq s'_{max})}{t^{\{s_{min},s_{max}\}} <: t^{\{s'_{min},s'_{max}\}}}$$

Finally, we give the rule for weakening via the subtype relation, which is used with S-SIZED, above. Notice the constraint set κ grows to include the sizing relationship between τ_1 and τ_2. If one were to expand the sizing pair to include more dimensions/aspects of type size, then the S-SIZED and T-WEAKEN constraints would be augmented to support the new sizing logic (Section 7 has more on this).

$$\text{(T-WEAKEN)}\ \frac{\Gamma \vdash \mathtt{e} : \tau_1\ \$\ c, \kappa \qquad \tau_1 <: \tau_2}{\Gamma \vdash \mathtt{e} : \tau_2\ \$\ c, \kappa \cup \kappa_2}$$

$$\text{where } \kappa_2 = \{\mathtt{minsize}(\tau_2) \leq \mathtt{minsize}(\tau_1),\ \mathtt{maxsize}(\tau_2) \geq \mathtt{maxsize}(\tau_1)\}$$

For some insight to non-type theorists: the above does not state that a smaller image may be substituted in place of a larger image, or vice-versa, but rather conveys the loosening of constraints when determining valid forms of input. The minimum size represents the largest size that we can guarantee, and the maximum size represents the smallest size that we can guarantee, thus decreasing the minimum size and/or increasing the maximum size are the only safe operations to allow for the subtype relation.

Notice there is a substantial difference between say, padding a zero in the case of substituting an integer for a real number and padding extrapolated content in the case of filling additional pixels via interpolation methods. Similarly, the reader might reason that images are similar to records of records and wish to provide the subtype relation in the opposite direction (ignoring those fields that are not necessary), however the analogy to records is erroneous: larger records contain extraneous, unrelated data while disregarding pixels in an image discards potentially useful data points (not unlike attempting to coerce a real number to an integer by rounding). As both image interpolation and scaling are potentially lossy operations, they should only be applied when explicitly requested by the programmer (a situation analogous to traditional type coercion via casting functions).

4.3 Conditionals

As an illustration of the construction of constraints (κ) and costs (c), the typing rule for a conditional (*i.e.*, if-then-else) is presented below. The cost function that we currently use is an upper bound of computational work. The equation for computing the cost of a conditional is the well-accepted bound of the cost of the predicate plus the larger of the two branches' costs. For the constraint set, we take the union of all sizing constraints (the predicate and both branches) even if one branch is ultimately unreachable.

(T-IF)

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathit{Bool}\,\$\,c_1, \kappa_1 \quad \Gamma \vdash \mathtt{e}_2 : \tau\,\$\,c_2, \kappa_2 \quad \Gamma \vdash \mathtt{e}_3 : \tau\,\$\,c_3, \kappa_3}{\Gamma \vdash \mathtt{cond\ e_1\ e_2\ e_3} : \tau\,\$\,c_0, \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

$$\text{where } c_0 = 1 + c_1 + \mathtt{max}(c_2\ c_3)$$

In the (common) event where terms of the conditional branches share a common base type but differ in size annotation, an application of weaken is used to relax the bounds on either side to meet at the lower minimum size and larger maximum size. If either branches' type is annotated with a size variable, each branch may be weakened using the subtype relationship to a new, common size variable for the conditional. Example A.1 in the Appendix portrays exactly this scenario: Two images (or expressions of type image) that have different, yet unknown sizes that are supplied as the branches of a conditional, and T-WEAKEN is applied to each prior to the application of the conditional to arrive at new size variables.

4.4 Constraints and Costs for Prim-Ops

Our type system treats all STEP Opcodes as primitive operations. The SNBENCH engineers who create their own opcodes are expected to define the cost function and type annotation constraints as a part of the Opcode's definition (*i.e.*, an extended type signature). The `lcost()` function (analogous to the `latentcost()` function defined in [17]) returns the discretized computational cost of each opcode as an equation of the size of its input. For example, the complexity of finding a face in an image is expected to be a function of the total number of pixels in the image. Our `lcost()` function diverges from the use in [17] insofar as we use not only the upper-bound cost of the function itself, but also include the derived type size bounds to better reflect the cost of computation (*i.e.*, produce a tighter size bound).

Two examples of primitive operations that manipulate images are included in Listing 4.4. The `lcost()` functions and sizes in this section have been contrived to ease the presentation and readability. For example, the face counting opcode (`facct`) requires that its input be in the size range of 320 to 1024, using image widths as a size rather than the actual total numbers of pixels (which would be in the millions of pixels). The definitions of these opcodes implicitly combine an aspect of T-WEAKEN, by using size ranges (rather than single values) in their constrained size variables.

The other example presented is the resample Opcode. A resampling operation alters the number of pixels in the image and in that respect is analogous to an explicit type casting operation to alter the size. While a resampling operation does increase the number of pixels in an image, it does not improve image *quality*. We will return to this issue in Figure 2. It has no explicit values for the size variables of its input, and its only constraint is that the input be of some positive size (greater than zero).

Although the two examples given place sizing constraints on image data, we could easily imagine operational constraints being placed on all data types. For example, the output of a `facct` operation is an Integer with sizes that range from a minimum of zero to a maximum that could be expressed as a function of the resolution of the input image.

4.5 Flowtypes

Finally we introduce a new function whose sole purpose is to inject run-time constraints (a Flowtype in SNBENCH nomenclature). The deadline flowtype indicates that an expression (its argument) has an explicit completion deadline. In terms of the typing formalism, the typing rule (shown below) augments the constraint set to include that the computational cost of the expression should be less than (or equal to) the specified deadline. The example given in Section 1 uses the function `period()` which would be implemented as syntactic sugar using `deadline()`.

$$\text{(T-DUE)}\ \frac{\Gamma \vdash \mathtt{n} : \mathit{Int}^{\{n,n\}}\,\$\,0, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau\,\$\,c_2, \kappa_2}{\Gamma \vdash \mathtt{deadline\ n\ e_2} : \tau\,\$\,c_2, \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

$$\text{where } \kappa_3 = \{c_2 \leq n\}$$

5. The Formal System in Practice

In this section several examples are constructed as an illustration of the use of the type system.

5.1 Inferring Optimal Image Resolution

In practice, a user may describe a computation that makes use of an image, yet the image itself is not part of the desired output. In such cases you would not expect the programmer to provide a specific resolution (size bound) for images that are used as a part of a larger computation. For example, a user who wishes to determine whether or not a light is on in a particular office is interested in a boolean result, not the intermediate image used to generate this result. Hardware image sensors (cameras) are able to capture images in a range of possible resolutions, and our type system can use its size constraint system to suggest an optimal resolution, a range of feasible resolutions, or indicate that there is no feasible solution for the program as specified.

$$\text{(T-FACECT)}\quad \frac{\mathit{type}(\texttt{facect}) = \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Int}^{\{n_1,n_2\}} \qquad \mathit{constr}(\texttt{facect}) = \kappa_1}{\texttt{facect} : \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Int}^{\{n_1,n_2\}}\ \$\ 0, \kappa_1}$$

$$\begin{aligned} \kappa_1 &= \{r_1 \geq 320, r_2 \leq 1024\} \\ \mathit{latentcost}(\texttt{facect}, \tau) &= (\mathit{maxsize}(\tau)/2) \end{aligned}$$

$$\text{(T-RESAMPLE)}\frac{\mathit{type}(\texttt{resample}) = \mathit{Int}^{\{n_1,n_2\}} \times \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Img}^{\{r_3,r_4\}} \quad \mathit{constr}(\texttt{resample}) = \kappa_1}{\texttt{resample} : \mathit{Int}^{\{n_1,n_2\}} \times \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Img}^{\{r_3,r_4\}}\ \$\ 0, \kappa_1}$$

$$\begin{aligned} \kappa_1 &= \{n_1 > 0, r_1 > 0, r_3 = r_1 * n_1, r_4 = r_2 * n_2\} \\ \mathit{latentcost}(\texttt{resample}, \tau) &= (\mathit{maxsize}(\tau)/8) \end{aligned}$$

Figure 2. The typing rule for the face-count and resample operations (*i.e.*, prim-ops). The type signature of these functions includes constraints and costs that enable the type system to construct the constraints and costs of larger, composite expressions.

Example 1:

This first example shows how the type system can be used to derive the resolution requirements from a code fragment, and how those constraints can be used to limit the selection of an image sensor at the time of dispatch.

```
(* face counting within an image from an
  image sensor with no explicit size bounds *)

 let IMG = get(sensor(IMAGE,ANY)) in
   facect(IMG)
```

While the complete derivation is presented in the appendix, the result of the derivation is presented here.

$$\frac{\vdots}{\texttt{facect(img)} : \mathit{Int}^{\{n_1,n_2\}}\ \$\ c_2, \kappa_1 \cup \kappa_2}$$

$$\begin{aligned} c_1 &= 1 + \mathit{lcost}(\texttt{get}, r_2) = 1 + (r_2/8) \\ c_2 &= 1 + c_1 + \mathit{lcost}(\texttt{facect}, r_2) = 1 + c_1 + r_2/2 \\ \kappa_1 &= \{r_1 \geq \texttt{r}_{min}, r_2 \leq \texttt{r}_{max}\} \\ \kappa_2 &= \{r_1 \geq 320, r_2 \leq 1024\} \end{aligned}$$

Solving the constraints for r_1 and r_2 (we minimize r_1 and maximize r_2 subject to the constraints given above, we determine the simple constraints that the resolution of the image to manipulate, and thus the sensor itself, fall in the range $[320, 1024]$. Thus the SSD may allocate any available sensor that can produce images within this range of resolutions. A camera that can capture images at resolutions ranging from 1024 to 4096 is valid for this service fragment (provided it samples at 1024), as is a camera that can capture images at the range from 320 to 512.

Example 2:

This second example builds on the first, this time counting faces within a *resampled* image. As before, the original image is captured from an image sensor with no explicit size bounds. The result of resampling is an image with four times as many pixels, and the face count opcode will perform more work as a result. The face count operation is not aware of the resolution of the original image, or the fact that the additional pixels offer no more information than the original (motivating a notion of image *quality* in Section 7).

```
(* face counting within a resized image from an
  image sensor with no explicit size bounds *)

 let IMG = get(sensor(IMAGE,ANY)) in
   facect(resample(4,IMG))
```

The costs and constraints that result from the derivation are presented below.

$$\frac{\vdots}{\texttt{facect(resample(4, img))} : \mathit{Int}^{\{n_3,n_4\}}\ \$\ c_3, \kappa_4}$$

$$\begin{aligned} c_1 &= 1 + \mathit{lcost}\,(\texttt{get}, r_2) = 1 + (r_2/8) \\ c_3 &= \mathit{lcost}\,(\texttt{facect}) + \mathit{lcost}\,(\texttt{resample}) + c_1 \\ \kappa_1 &= \{r_1 \geq \texttt{r}_{min}, r_2 \leq \texttt{r}_{max}\} \\ \kappa_2 &= \{n > 0, r_1 > 0, r_3 = r_1 * 4, r_4 = r_2 * 4\} \\ \kappa_3 &= \{r_3 \geq 320, r_4 \leq 1024\} \\ \kappa_4 &= \{n = 4\} \cup \kappa_1 \cup \kappa_2 \cup \kappa_3 \end{aligned}$$

We obtain that $r_1 * 4 \geq 320 \Rightarrow r_1 \geq 80$ and $r_2 * 4 \leq 1024 \Rightarrow r_2 \leq 256$. Minimizing for r_1 and maximizing for r_2 gives the resolution range [80,256] required of the sensor to be allocated for this fragment.

Bear in mind that the minimum resolution in a range of resolutions is not always the most desirable value; while the minimum will consume the least computational resources, it may do so at the expense of computation confidence (*e.g.*, it may not be possible to detect all the faces in an image if the resolution is too small). In other scenarios, the use of the maximum resolution (that the resources available can accommodate) may be an unnecessary overhead. The processing overhead is easily measured by computing the cost function (c_2, in the above) for both the maximum and minimum feasible resolutions.

5.2 Computational Cost and Size Bounds

The worst case computational cost is given as a result of our type system as the first argument after the $ in a typing judgment. This is an approximation and could be adjusted/calibrated as a function to estimate actual execution times on various physical resources. The worst-case computational bounds provided by our system provide one static-time validation mechanism to determine if explicit deadlines or other run-time constraints (given Flowtypes) cannot be met as specified. As presented, the `deadline` opcode relies on this unadjusted approximation of computation time.

Example 3:

Building on Example 1, this example performs a face counting operation on an image with no explicit size bounds, yet the computation has an explicit deadline.

```
(* counting faces from an arbitrary image sensor.
   The computation has an explicit deadline *)

 let IMG = get(sensor(IMAGE,ANY)) in
  deadline(322,facect(IMG))
```

Again, only the result of the derivation is presented here. The complete derivation is available in the appendix.

$$\frac{\vdots}{\texttt{deadline}\,(322, \texttt{facect}(\texttt{img})) : \mathit{Int}^{\{n_1,n_2\}}\ \$\ c_2, \kappa_u}$$

$$\begin{array}{rcl} c_1 & = & 1 + \mathit{lcos}(\texttt{get}, r_2) = 1 + (r_2/8) \\ c_2 & = & 1 + c_1 + \mathit{lcost}(\texttt{facect}, r_2) = 1 + c_1 + r_2/2 \\ \kappa_1 & = & \{r_1 \geq \mathtt{r}_{min}, r_2 \leq \mathtt{r}_{max}\} \\ \kappa_2 & = & \{r_1 \geq 320, r_2 \leq 1024\} \\ \kappa_3 & = & \{c_2 \leq 322\} \\ \kappa_u & = & \{n = 322\} \cup \kappa_1 \cup \kappa_2 \cup \kappa_3 \end{array}$$

In this example the valid range for capture is no longer $[320, 1024]$. Solving for r_1 and r_2 the range is now limited to $[320, 512]$ (as larger values of r_2 would exceed the constraint imposed by κ_3).

In addition to the static verification of Flowtypes, the computed cost may be leveraged in other ways as well. The key responsibility of the Service Dispatcher component of SNBENCH is to partition a task across physical resources (if there is insufficient computing resources available on a single node to accommodate the entire task). The worst case computational cost, as presented, provides us with an initial metric to guide the allocation of physical computing nodes and sensory resources for a given task (and its constituent subtasks). Similarly, for any given task or sub-task we can look at the upper bound size annotation on its type information to determine the potential network overhead associated with partitioning the larger task at that point.

6 Implementation Details

In its current incarnation, the type system described in this document is presented as part of the SNBENCH program development tool chain. The implementation is programmed in Java and makes use of the open-source JavaCC (Java Compiler Compiler) project [18]. The type checker is automatically invoked when compiling our high-level language (SNAFU) programming language to STEP. Every constraint that is added to the constraint set (κ) also includes the line number in the SNAFU source code that caused this constraint to be added. In this way, should unification fail, the tool can point the programmer to the problematic line in their native source language (reporting an error to a SNAFU programmer in STEP would be not unlike reporting an error by referring to the bytecode rather than the original Java source code).

As the implementation is entirely modular and checks STEP code rather than SNAFU code, nothing prevents the use of the type checker *apart from* the SNAFU compilation process (*i.e.*, on the Service Dispatcher, at the time of task submission). However (1) the type checker implementation includes several hooks to track line numbers within SNAFU code as a convenience to the users of SNAFU and (2) SNAFU is eminently more readable than STEP. It is our intent that the type checker will be invoked by the compilers of other future high-level languages in the SNBENCH.

While the implementation of the type checking engine includes some form of support for all the major constructs in the STEP programming language, there are some limitations to this support. For STEP constructs or prim-ops for which complete type annotation/constraint is lacking, heuristics are used to bound cost. In the worst case, for those opcodes which lack complete size constraint annotation, unsized type checking is used rather than the size annotated type checking.

To solve the constraint sets we invoke the GNU Linear Programming Toolkit (GLPK) [2]. GLPK can be used to solve a system of constraints for linear programming and mixed integer programming. The decision to use GLPK

is based on project maturity, community support and API availability. At present we emit our sizing constraints in the GNU MathProg language and invoke the GLPK via an automated script, though nothing (other than time) precludes finer integration via the GLPK Java Interface [5]. The use of a linear solver limits the complexity of constraints that can be represented, however we have plans to integrate a Quadratic solver in the near future.

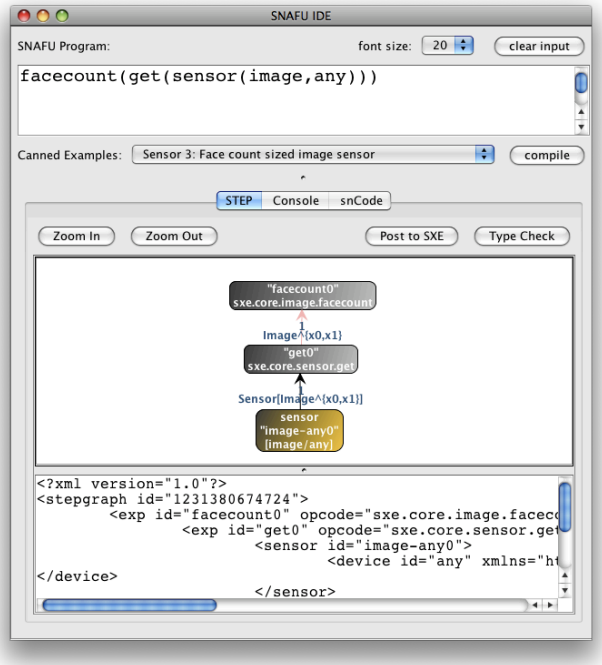

Figure 3. A screen shot from the SNAFU development environment that shows the successful type checking of a STEP program. Results generated from the implementation of the type checker are given to the user graphically.

7 Future Work

Additional Type Annotations

In this paper we have presented upper and lower size bound data type annotation, yet other useful annotations exist and can be easily integrated into this type system by extending the existing annotation pair to a tuple or keyed set and defining the desired subtyping relation and expanded type signatures.

One such example is the notion of image quality, which is different from data size. Data *size* is used to bound the operational requirements of a function (including those that manipulate images), whereas image *quality* speaks to the valid data in the image. As far as operational/functional correctness is concerned a resized image is operationally valid, however with respect to the desired programmatic output, a smaller image that has been resized to a larger resolution is not truly interchangeable with an image captured at a larger resolution. When an image is resized (*e.g.*, via scaling or resampling, say) the minimum size (resolution) of the image no longer reflects the original number of data points (we call this "quality").

We could easily support a notion of an image's quality within our type annotations by adding a dimension for the "lowest" value that we have ever seen for a lower size bound (recall the definition of T-RESAMPLE increases both the upper and the lower size bound). This value could distinguish between a true high resolution image and data that has been (perhaps foolishly) up-cast or coerced to satisfy a function's minimum size constraints. Quality also has a well established meaning with respect to numerical data as well, and might be defined to reflect potential for rounding errors, data accuracy, *etc*. Certainly other image and video related aspects could be tracked as well, including color-depth for images, frame rates for video, and so on.

Applications to Image Pyramids

Image Pyramids [11],[1] are a well established technique in the field of image processing. The technique involves maintaining multiple copies of the same image at different resolutions (the a hierarchy of resolutions form a logical pyramid) such that an image of the "ideal" resolution can be used for processing depending on the needs of the specific processing function. The beneficial applications of our formalism to this image processing community is potentially two-fold. (1) We can extend the type system to include an image pyramid as a first class type and extend the annotations to support the list of resolutions available in the pyramid. (2) Static analysis of the image processing flow can tell us precisely which resolutions need to be kept in the pyramid and which can be removed. In the latter case, the potential benefit of SNBENCH and sized typing is quite significant as it might be possible to remove the pyramid entirely. For distributed computations, we can split the pyramid into individual image instances based on the analysis of size/use and ensure that we are passing as few copies of the image (inside the pyramid) and as few copies of the pyramid itself, as possible.

Costs and Sizes for New Opcodes

The operational correctness of the type system is contingent on the presence of accurate size, cost and constraint data for Opcodes (primitive operators). SNBENCH provides a facility by which new Opcodes may be added to a service library quickly and easily, through the implementation of a simple Java interface. At present the type system maintains an embedded definition for the latent costs and size constraints of the current Opcode library, however for the sustainability of the type system, it is essential that these definitions are provided by the Opcode authors and automatically extracted directly from the Opcode definitions. Beside changing the Opcode implementation interface, the

type system must also change to import the rules from the Opcode library directly. There is also a concern that Opcode authors might specify very weak size constraints and very high costs (possibly lacking the knowledge to do so correctly) and that the end result is a type system that is only as strong as its weakest link. Any future work that would automatically extract this information from an Opcode implementation would be a fantastic solution to this problem (and others).

Different Models of Cost

At present our type system provides a single model of computational cost, a worst-case upper bound. Just as there is the possibility for multiple size annotations or dimensions of types, there is also an opportunity to provide multiple cost metrics based on these additional dimensions. Other cost models would be possible, including a "minimum" computational cost (or optimal case), an average computational cost, a financial cost, and a total memory utilization cost. The most elegant seeming approach would be to support multiple, user defined functional costs, such that multiple costs appear on the right hand side of the $, and each would be computed from user-provided functions that manipulate the annotations available in the typing rules.

8 Conclusions

In this paper, we have presented our formal type system for multi-dimensional sized types for a functional-style, sensing-centric domain specific language. Unlike other sized type systems our work tracks both an upper and lower size bound for data, defines a logical subtype relation for images capable of bounding computation, maintaining functional correctness, and deduce feasible data sizes from implicit and explicit constraints within program fragments. We presented this system and have provided examples that illustrate the use of the type system. We are confident in the many potential uses for this formalism to the image processing and Sense-and-Respond communities.

References

[1] F. Ackermann and M. Hahn. Image pyramids for digital photogrammetry. *Digital Photogrammetric Systems*, pages 43–59, 1991.

[2] Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia. GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, pages 1–26. Springer-Verlag, 2005.

[4] A. Bestavros, A. Bradley, A. Kfoury, and M. Ocean. SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications. In *IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets)*, October, 2005.

[5] Bjoern Frank. GNU Linear Programming Kit 4.8 Java Interface. http://bjoern.dapnet.de/glpk/.

[6] Boston University, CS Dept. Sensorium Research Homepage. http://www.cs.bu.edu/groups/sensorium/.

[7] Boston University, CS Dept. snBench Research Homepage. http://csr.bu.edu/snbench/.

[8] K. Hammond, C. Ferdinand, and R. Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Rev.*, 3(4):27–36, 2006.

[9] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL03*, pages 185–197. ACM Press, 2003.

[10] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM PoPL*, pages 410–423, 1996.

[11] W. Kropatsch. Image pyramids and curves. an overview. Technical report, Department for Pattern Recongition and Image Processing of the Institute of Automation, University of Technology, Vienna, Austria, 1991.

[12] H.-W. Loidl and K. Hammond. A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM TOPLAS*, pages 85–97, 1999.

[14] M. Ocean, A. Kfoury, and A. Bestavros. A Type System For Safe SN Resource Allocation. Technical Report BUCS-TR-2008-011, CS Dept., Boston University, June 14 2008.

[15] M. J. Ocean and A. Bestavros. Extending the snBench to support wireless intrusion detection. In *WiSec'08: The First ACM Conference on Wireless Network Security*, Alexandria, Virginia, March 2008. (Best Paper Award).

[16] M. J. Ocean, A. Bestavros, and A. J. Kfoury. SNBENCH: Programming and Virtualization Framework for Distributed Multitasking Sensor Networks. In *VEE '06: The 2nd International Conference on Virtual Execution Environments*, pages 89–99, New York, NY, USA, 2006. ACM Press.

[17] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

[18] Sriram Sankar, Metameta. Java Compiler Compiler (Java CC). The Java Parser Generator. http://javacc.dev.java.net/.

[19] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2-3):157–179, 2000.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[21] H. Xi. Applied type system (extended abstract). In *LNCS 3085*, pages 394–408. Springer-Verlag, 2004.

[22] H. Xi and R. Harper. A dependently typed assembly language. *ACM SIGPLAN*, 36(10):169–180, 2001.

A. Sample Derivations

The derivations presented here refer to typing rules that do not appear in this report, yet are included here to give the reader a sense of what is involved in the formal system.

A.1 Conditionals and Subtypes

$$\dfrac{\dfrac{\vdots}{\texttt{b} : \mathit{Bool}\ \$ c_0, \kappa_0} \quad \dfrac{\texttt{i}_1 : \mathit{Img}^{\{r_1,r_2\}}\ \$\ c_1, \kappa_1}{\texttt{i}_1 : \mathit{Img}^{\{r_5,r_6\}}\ \$\ c_1, \kappa_1 \cup \kappa_1'}(\text{T-WEAKEN}) \quad \dfrac{\texttt{i}_2 : \mathit{Img}^{\{r_3,r_4\}}\ \$\ c_2, \kappa}{\texttt{i}_2 : \mathit{Img}^{\{r_5,r_6\}}\ \$\ c_2, \kappa \cup \kappa_2'}(\text{T-WEAKEN})}{\texttt{cond b i}_1\ \texttt{i}_2 : \mathit{Img}^{\{r_5,r_6\}} \$\ c_0 + \mathit{max}(c_1, c_2), \kappa_0 \cup \kappa_1 \cup \kappa_1' \cup \kappa_2 \cup \kappa_2'}(\text{T-COND})$$

where:

$$\kappa_1' = \{r_5 \leq r_1, r_6 \geq r_2\}$$
$$\kappa_2' = \{r_5 \leq r_3, r_6 \geq r_4\}$$

A.2 Derivation for Example 1

$$\mathcal{D}_0 = \dfrac{\mathit{type}(\texttt{get}) = \mathit{Sensor}\ \tau \rightarrow \tau \quad \dfrac{}{\texttt{image} : \mathit{Sensor}\ \mathit{Img}^{\{r_1,r_2\}}\ \$\ 0, \kappa_1}(\text{T-IMAGESENSOR})}{\texttt{get(image)} : \mathit{Img}^{\{r_1,r_2\}}\ \$\ c_1, \kappa_1}(\text{T-SENSORREAD})$$

$$\mathcal{D}_1 = \dfrac{\dfrac{\mathit{type}(\texttt{facect}) = \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Int}^{\{n_1,n_2\}} \qquad \mathit{constr}(\texttt{facect}) = \kappa_2}{\texttt{facect} : \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Int}^{\{n_1,n_2\}}\ \$\ 0, \kappa_2}(\text{T-FACECT}) \qquad \mathcal{D}_0}{\texttt{facect(get(image))} : \mathit{Int}^{\{n_1,n_2\}}\ \$\ c_2, \kappa_1 \cup \kappa_2}(\text{T-OPAPPLY})$$

$$\begin{aligned} c_1 &= 1 + \mathit{lcost}(\texttt{get}, r_2) = 1 + (r_2/8) \\ c_2 &= 1 + c_1 + \mathit{lcost}(\texttt{facect}, r_2) = 1 + c_1 + r_2/2 \\ \kappa_1 &= \{r_1 \geq \texttt{r}_{min}, r_2 \leq \texttt{r}_{max}\} \\ \kappa_2 &= \{r_1 \geq 320, r_2 \leq 1024\} \end{aligned}$$

A.3 Derivation for Example 3

$$\dfrac{\dfrac{}{322 : \mathit{Int}^{\{n,n\}}\ \$\ 0, \kappa_0}(\text{T-INT}) \qquad \mathcal{D}_1}{\texttt{deadline 322 facect(get(image))} : \mathit{Int}^{\{n_1,n_2\}}\ \$\ c_2, \kappa_0 \cup \kappa_1 \cup \kappa_2 \cup \kappa_3}(\text{T-DEADLINE})$$

$$\begin{aligned} c_1 &= 1 + \mathit{lcost}(\texttt{get}, r_2) = 1 + (r_2/8) \\ c_2 &= 1 + c_1 + \mathit{lcost}(\texttt{facect}, r_2) = 1 + c_1 + r_2/2 \\ \kappa_1 &= \{r_1 \geq \texttt{r}_{min}, r_2 \leq \texttt{r}_{max}\} \\ \kappa_2 &= \{r_1 \geq 320, r_2 \leq 1024\} \\ \kappa_0 &= \{n = 322\} \\ \kappa_3 &= \{c_2 \leq 322\} \end{aligned}$$