A Type System For Safe SN Resource Allocation

Michael Ocean Assaf Kfoury Azer Bestavros
Computer Science Department
Boston University
Boston, MA 02215

Technical Report: BUCS-TR-2008-011
June 14, 2008

Abstract

snBench is a platform on which novice users compose and deploy distributed Sense and Respond programs for simultaneous execution on a shared, distributed infrastructure. It is a natural imperative that we have the ability to (1) verify the safety/correctness of newly submitted tasks and (2) derive the resource requirements for these tasks such that correct allocation may occur. To achieve these goals we have established a *multi-dimensional* sized type system for our functional-style Domain Specific Language (DSL) called Sensor Task Execution Plan (STEP). In such a type system data types are annotated with a vector of size attributes (*e.g.*, upper and lower size bounds). Tracking multiple size aspects proves essential in a system in which Images are manipulated as a first class data type, as image manipulation functions may have specific minimum and/or maximum resolution restrictions on the input they can correctly process.

Through static analysis of STEP instances we not only verify basic type safety and establish upper computational resource bounds (*i.e.*, time and space), but we also derive and solve data and resource sizing constraints (*e.g.*, Image resolution, camera capabilities) from the implicit constraints embedded in program instances. In fact, the static methods presented here have benefit beyond their application to Image data, and may be extended to other data types that require tracking multiple dimensions (*e.g.*, image "quality", video frame-rate or aspect ratio, audio sampling rate). In this paper we present the syntax and semantics of our functional language, our type system that builds costs and resource/data constraints, and (through both formalism and specific details of our implementation) provide concrete examples of how the constraints and sizing information are used in practice.

† This research was supported in part by a number of NSF awards, including CISE/CSR Award #0720604, ENG/EFRI Award #0735974, CISE/CNS Award #0524477, CNS/NeTS Award #0520166, and CNS/ITR Award #0205294.

1 Introduction

Motivation

The Sensor Network WorkBench (snBench) 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 novice users to compose these services we provide our users with a functional-style Domain Specific Language (DSL) for specification, called STEP (Sensor Task Execution Plan).[1] 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 architecture and run-time components have been published elsewhere ([OBK06]), in this paper we present the static analysis techniques that we have developed to provide safety and resource constraint extraction on our sensing-centric STEP language. 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 recognize 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 not only bound costs for memory and computation, but 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

[1] We actually provide other, high-level languages that are compiled down to STEP as our common Instruction Set Architecture.

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, take an image from "any" camera and
try to detect motion.  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. While the explicit periodicity function (or as well call it, Flowtype) **period** indicates that this expression must run every 100 milliseconds. Thus there is another constraint on the resolution; the resolution must also be low enough to allow computation 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 can not be met given current available resources).

In this paper we present our type system as applied to a subset of our domain specific language, give details of its implementation, and work through concrete examples of the system in use. The organization of the rest of this paper is as follows: Section 2 provides a brief, high-level overview of snBench, Section 3 provides the syntax of our DSL and the static and dynamic semantics of our type system, and Section 4 applies this formalism to some examples to show the system in action as well as giving an overview of our implementation. In Section 5 we indicate some of the many possible future directions for this work.

Related Work

Bounding the execution time of programs and program fragments is a well-established problem in computing. There was a large interest in applying custom type systems to domain specific languages in the late nineties (*e.g.*, the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999).

Indeed, Our work has been largely inspired by existing works that aim to solve execution time bounding via an upper bound size annotation on types within a type system. These works include Static Dependent Costs [RG94], Sized Type Systems [HPS96], and Sized Time Systems [LH96]. All establish a formal type system that has been annotated with upper size bounds on datatypes to estimate an upper bound on execution time and memory requirements given input size. We intended to use these techniques (with small adjustments to support STEP) in order to verify basic type safety and extract execution and memory bounds to guide resource allocation and scheduling within snBench. The operating environment of snBench is intrinsically distributed and time dependent, yet it seemed that these existing works should have been able to be ported more or less directly ([HFH06], for example, is the current incarnation of the Sized Time System, and is aimed directly at the real-time embedded sensing community). However, this was not the case.

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. Thus our needs began to diverge almost immediately. In the existing works size annotation has nothing to do with functional correctness, moreover we recognized a need to track a lower size bound (annotation) in addition to the upper size bound. From the need to add the additional lower size bound we have established a system in which the size annotations are *multi-dimensional*; while the formalism described in section 3 only includes a lower and upper size bound, Section 5 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 such 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.

It is also worth noting that our language includes a slightly unusual let semantic which is advantageous in time dependent programming environments. While several functional languages include this same let behavior in their implementation, in time-independent operating environments implementation of the let discipline in this way does not influence the result of the computation, but rather is provided as an optimization. In our environment, this is not the case. Thus, we define our let semantic formally and include it in our proofs for soundness and completeness.

2 snBench Overview

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 [BBKO05] and implementation details may be found in [OBK06].

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 [Bos]. 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. We liken the support that snBench extends to a Sensor Network 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.

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 deploy (bind) tasks on to available SXEs. A graphical representation of this end-to-end support is shown in Figure 1.

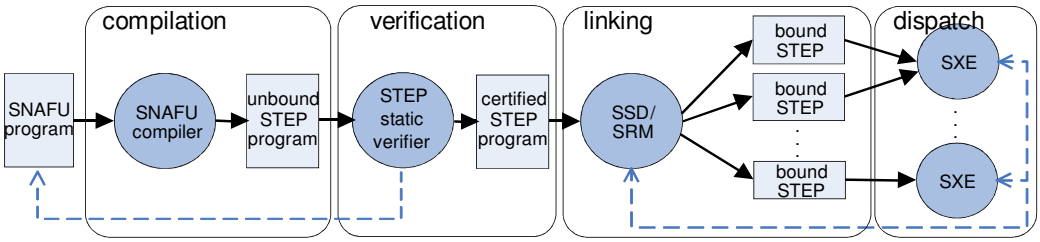

Figure 1: The SN program life-cycle as enabled by snBench. Rectangles represent data, circles represent tasks/processes, and the dashed lines represent control communication (dependency).

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 dependency with the nodes of a STEP graph representing the atomic computation and sensing operations and edges representing data flow. In execution, demand for evaluation pushed 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.

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 and ensure that the resources required by the tasks are currently available in the system. 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). [2] It is our type system that provides this certification and the resource requirements it extracts are essential to enabling the Service Dispatcher to correctly allocate resources (SXEs).

3 Formalism for "Core" STEP

In this section we present the formal logic that underlies the verification component described above. The type system we present in this Section supports a subset of the complete STEP programming language; we call this subset "Core" STEP. We discus the challenges of supporting the few remaining STEP components in Section 5.

Readers who are actively familiar with the project may note that what is presented as STEP appears to be STEP's high-level, functional sibling SNAFU. In fact, SNAFU is a largely a convenience wrapper for STEP, including some additional syntactic sugar (which we do not address in this paper).

In Core STEP we formally present only one let form. As we show later, this let behaves in a way that is temporally interesting. The Complete STEP supports other, traditional let forms (*i.e.*, lazy and eager), yet we omit them from the Core formalism as they are well-known and relatively less interesting in our domain.

3.1 Syntax of Expressions

Below we define the syntax of the valid expression forms of Core STEP.

[2] Certification could happen entirely on the client side in an environment in which the user/service composer can not forge certification output.

e	::=		***expressions***
		v	*value*
		x	*variable*
		`cond` e e e	*conditional (if-then-else)*
		`let` $\{x_i{=}\mathrm{e}_i\}^{i\in 1..n}$ `in` e	*let binding*
		`get` e	*read a sensor*
		op e	*opcode/primitive operation*
		`trigger` $\{x_i{=}\mathrm{e}_i\}^{i\in 1..n}$ e e	*construct for repetition*
v	::=		***values***
		$0 \mid 1 \mid 2 \mid \ldots$	*integer*
		`true` | `false`	*boolean*
		i	*image*
		`time` | `image`	*sensor*
op	::=	$op_1 \mid op_2 \mid op_3 \mid \ldots$	***opcodes***

3.2 Preliminary Definitions

The evaluation (dynamic semantics) of expressions take the general form:

$$\mathtt{e} \mid \nu \mid t \rightarrow \mathtt{e}' \mid \nu' \mid t+1$$

Where ν is a special variable store required for our let semantic, and t is a discretized time that increments per evaluation step (*e.g.*, computation count, clock tick). We read this in English as " the expression `e` and its variable store ν take an evaluative step to the new expression $\mathtt{e}'$ with new variable store ν'."

3.2.1 Definition: The Variable Store ν:

The evaluation of the STEP language is time dependent insofar as values that are read from the sensing environment may change over time. Thus, *when* a function is evaluated will directly effect the value retrieved (specifically *via* the sensor reading function, `get`). Put differently, the let semantic we provide has a direct impact on the values retrieved for a variable (*e.g.*, an eager let retrieves a reading at time 0, a typical lazy or by-need let retrieves a new reading at every time the substituted variable is encountered). To best accommodate the needs of STEP and this temporal dependence, we define our default let-binding to be a hybrid approach; we offer a deferred evaluation (*i.e.*, by need) that also offers the reuse provided by eager evaluation (*i.e.*, with caching).

To achieve the desired result our let-binding construct manipulates a variable store ν that stores the mapping of variables to unreduced expressions which will be reduced within the environment itself when the variable is encountered (by need). Should the same variable be encountered again further in the expression, the mapping in ν will later point to the fully reduced value and will therefore return that previously computed value (with caching).

We define the store, ν below, whose domain ranges over variable symbols, each pointing to valid STEP expression (either an unreduced expression or a value).

$$\begin{aligned} \nu &= \{x_1 \mapsto \mathtt{e}_1, x_2 \mapsto \mathtt{e}_2, \ldots, x_n \mapsto \mathtt{e}_n\} \\ \mathit{Domain}(\nu) &= \{x_1, \ldots, x_n\} \\ \nu(x_i) &= \mathtt{e}_i \end{aligned}$$

3.2.2 Definition: The free-variable function *FV*

FV calculates the free-variables in an expression ($\mathtt{e}$) or expression and store pairing ($\mathtt{e} \mid \nu$).

$$\begin{aligned} \mathit{FV}(\mathtt{v}) &= \varnothing \\ \mathit{FV}(x) &= \{x\} \\ \mathit{FV}(\mathtt{cond\ e_1\ e_2\ e_3}) &= \mathit{FV}(\mathtt{e}_1) \cup \mathit{FV}(\mathtt{e}_2) \cup \mathit{FV}(\mathtt{e}_3) \\ \mathit{FV}(\mathtt{let\ } \{x_i = e_i\}^{i \in 1..n} \mathtt{\ in\ e}) &= (\bigcup_{i=1}^{n} \mathit{FV}(\mathtt{e}_i) \cup \mathit{FV}(\mathtt{e})) - \{x_1 \ldots x_n\} \\ \mathit{FV}(\mathtt{get\ e}) &= \mathit{FV}(\mathtt{e}) \\ \mathit{FV}(\mathit{op}\ \mathtt{e}) &= \mathit{FV}(\mathtt{e}) \\ \mathit{FV}(\mathtt{trigger\ } \{x_i = e_i\}^{i \in 1..n}\ \mathtt{e}_{n+1}\ \mathtt{e}_{n+2}) &= (\bigcup_{i=1}^{n} \mathit{FV}(\mathtt{e}_i) \cup \mathit{FV}(\mathtt{e}_{n+1}) \cup \mathit{FV}(\mathtt{e}_{n+2})) - \{x_1 \ldots x_n\} \\ \mathit{FV}(\mathtt{e} \mid \nu) &= (\bigcup \{\mathit{FV}(\nu(x)) \mid x \in \mathit{Domain}(\nu)\} \cup \mathit{FV}(\mathtt{e})) - \mathit{Domain}(\nu) \end{aligned}$$

3.2.3 Definition: expression and store pair closure

We say $\mathtt{e} \mid \nu$ is **closed** if and only if $\mathit{FV}(\mathtt{e} \mid \nu) = \varnothing$.

3.3 Dynamic Semantics

In this section we present the evaluation rules (dynamic semantics) for the various constructs (*i.e.*, syntactic forms) of the language.

3.3.1 Conditional

$$\text{(E-IfTrue)}\quad \mathtt{cond\ true\ e_2\ e_3} \mid \nu \mid t \rightarrow \mathtt{e_2} \mid \nu \mid t+1$$

$$\text{(E-IfFalse)}\quad \mathtt{cond\ false\ e_2\ e_3} \mid \nu \mid t \rightarrow \mathtt{e_3} \mid \nu \mid t+1$$

$$\text{(E-If)}\quad \frac{\mathtt{e_1} \mid \nu \mid t \rightarrow \mathtt{e_1'} \mid \nu' \mid t+1}{\mathtt{cond\ e_1\ e_2\ e_3} \mid \nu \mid t \rightarrow \mathtt{cond\ e_1'\ e_2\ e_3} \mid \nu' \mid t+1}$$

3.3.2 Opcodes

We refer to primitive operators in STEP as Opcodes. Evaluation of opcodes is strict (arguments must be reduced to values before the opcode may be evaluated). Evaluation and typing rules for specific instances of opcodes (*e.g.*, `facedetect`, `resample`) are given in Section 4. The general form is presenter here.

$$\text{(E-Op1)}\quad \frac{\mathtt{e}_1 \mid \nu \mid t \;\rightarrow\; \mathtt{e}_1' \mid \nu' \mid t+1}{op\ \mathtt{e}_1 \mid \nu \mid t \rightarrow op\ \mathtt{e}_1' \mid \nu' \mid t+1}$$

$$\text{where } op \in \{op_1 \mid op_2 \mid op_3 \mid \ldots\}$$

$$\text{(E-OpApply)}\quad op\ \mathtt{v}_1 \mid \nu \mid t \;\rightarrow\; \mathtt{v}_2 \mid \nu \mid t+1$$

$$\text{where } \mathtt{v}_2 = \mathtt{Apply}(op\ \mathtt{v}_1)$$
$$\text{and } op \in \{op_1 \mid op_2 \mid op_3 \mid \ldots\}$$

3.3.3 Sensor reads (and the physical sensor environment $\mathcal{E}$)

As STEP is a sensing centric DSL, it is essential that we have the ability to read values from sensors that are embedded in the physical environment. We imagine a logical array that contains all the data that a sensor will produce at every discretized time interval. Reading a value from a sensor is analogous to extracting a value from this array indexed at the current time (t). We define an abstract matrix $\mathcal{E}$ to correspond to the physical sensing environment such that $\mathcal{E}_{i,j}$ is the reading (value) from sensor i at discretized time j. We define the function $f_{\mathcal{E}}(i,j)$ to extract the j^{th} value (corresponding with time j) from stream (sensor) i from $\mathcal{E}$.

We define the (strict) function `get` to extract a value from the sensing environment. In the simplified Core STEP we support only two types of sensors, a time sensor (`time`) and Image sensors (`image`).

$$\text{(E-Get1)}\quad \frac{\mathtt{e}_1 \mid \nu \mid t \;\rightarrow\; \mathtt{e}_1' \mid \nu' \mid t+1}{\mathtt{get}\ \mathtt{e}_1 \mid \nu \mid t \rightarrow\ \mathtt{get}\ \mathtt{e}_1' \mid \nu' \mid t+1}$$

$$\text{(E-GetApply)}\ \ \mathtt{get}\ \mathtt{v}_1 \mid \nu \mid t \rightarrow\ \mathtt{v}_2 \mid \nu \mid t+1$$

$$\text{where } \mathtt{v}_2 = f_{\mathcal{E}}(\mathtt{v}, t)$$
$$\text{and } f_{\mathcal{E}} : \mathit{Sensor}\ \tau \times \mathit{Int} \;\rightarrow \tau$$

The `time` sensor returns the number of evaluations (computations) since the beginning of the evaluation (*i.e.*, returns t).

$$\text{(E-GetTime)}\ \ (\mathtt{get\ time} \mid \nu \mid t) \rightarrow\ (t \mid \nu \mid t+1)$$

3.3.4 Let binding (by-need with caching)

Again, our let semantic is a hybrid approach that is deferred evaluation (*ala* lazy) coupled with evaluation re-use (*ala* by-value/eager). To facilitate this, when a let is encountered, its assignments are added directly to the store ν (*via* E-LetN) without evaluation. Expressions and instead are evaluated within ν when their variable is encountered elsewhere (*via* E-Var1 and E-Var2) including variables that point to other variables. Our let expression allows for simultaneous assignment; in E-LetN we assume that all terms x_i are assigned simultaneously and may have interdependency.[3]

Variable terms are subject to alpha renaming to avoid variable capture, *etc.*

$$\text{(E-LetN)}\quad \begin{array}{l}\texttt{let}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{in}\ \texttt{e}_{n+1} \mid \nu \mid t \\ \rightarrow\ \texttt{e}_{n+1} \mid \nu \cup \{x_1 \mapsto\ \texttt{e}_1, \ldots, x_n \mapsto\ \texttt{e}_n\} \mid t+1\end{array}$$

$$\text{(E-Var1)}\quad \frac{\texttt{e}_1 \mid \nu \mid t\ \rightarrow\ \texttt{e}_1' \mid \nu' \mid t+1}{(x \mid \nu \cup \{x \mapsto \texttt{e}_1\} \mid t) \rightarrow\ (x \mid \nu' \cup \{x \mapsto \texttt{e}_1'\} \mid t+1)}\quad x \notin \mathit{domain}(\nu) \cup \mathit{domain}(\nu')$$

$$\text{(E-Var2)}\quad (x \mid \nu \cup \{x \mapsto \texttt{v}\} \mid t) \rightarrow\ (\texttt{v} \mid \nu \cup \{x \mapsto \texttt{v}\} \mid t+1)$$

3.3.5 Triggers

STEP provides a `trigger` construct to specify 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). We also provide the ability to provide sequential let-bindings that behave as though they are within the scope of each trigger expression evaluation (*i.e.*, the let-term is recomputed on every expansion of the trigger) are are available to both branches of the trigger (a single let-binding result spans both trigger arguments).

We define the trigger's repetition recursively, via the conditional. For completeness we also define functionally degenerate trigger forms; expressions that specify a constant value for the trigger predicate produce a superfluous trigger expression (*i.e.*, `trigger true` $\texttt{e}_1$ always reduces to $\texttt{e}_1$ while `trigger false` $\texttt{e}_1$ would never proceed).

$$\text{(E-TriggerDeg)}\quad \begin{array}{l}\texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2 \mid \nu \mid t \\ \rightarrow \texttt{cond}\ \texttt{e}_1\ \texttt{e}_2\ (\texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2) \mid \nu \mid t+1\end{array}$$

$$\text{(E-TriggerDeg)}\quad \frac{\texttt{e}_2 \mid \nu \mid t \rightarrow \texttt{e}_2' \mid \nu' \mid t+1}{\texttt{trigger v e}_2 \mid \nu \mid t \rightarrow \texttt{cond v e}_2'\ (\texttt{trigger v e}_2) \mid \nu' \mid t+1}$$

The `letonce` expression shown is syntactic sugar for the expanded trigger expression, which builds a sequential set of variable assignments for which there is one one expansion per trigger iteration (alpha renaming ensures we are not using the same variable in every iteration). The binding is intentionally placed at the scope of both the predicate and the conclusion (*i.e.*, consequent) of the trigger term.

$$\texttt{letonce}\ \{x_1{=}\texttt{e}_1, \ldots, x_n{=}\texttt{e}_n\}\ \texttt{in trigger}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2} \equiv \texttt{trigger}\ \{x_1{=}\texttt{e}_1, \ldots, x_n{=}\texttt{e}_n\}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2}$$

[3] Put differently, the multiple assignments in E-LetN are *not* syntactic sugar for nested lets.

$$\text{(E-TriggerLet)}\quad \begin{array}{l} \texttt{trigger}\ \{x_1{=}\texttt{e}_1,\ldots,x_n{=}\texttt{e}_n\}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2} \\ \rightarrow\ \texttt{let}\ \{x_1{=}\texttt{e}_1,\ldots,x_n{=}\texttt{e}_n\}\ \texttt{in} \\ \quad \texttt{cond}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2}\ (\texttt{trigger}\ \{x_1{=}\texttt{e}_1,\ldots,x_n{=}\texttt{e}_n\}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2}) \end{array}$$

3.4 Syntax of Types

The syntax of types in Core STEP are given below.

t	::=		***base types***
		Int | *Bool* | *Img*	
τ_p	::=		***primitive types***
		$t^{\{s,s\}}$	*w/ size annotation*
τ	::=		***types***
		τ_p	*primitive type*
		Sensor τ_b	*sensor*
		$\tau\ \rightarrow\ \tau$	*opcode*
s	::=	$n \mid r$	***size annotation***

3.5 Static Semantics and Sizing of Types

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

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

Where t is base type (*e.g.*, *Int* , *Bool* , *Img*), $\{s_{min}, s_{max}\}$ is the *size* annotation for the type (s_{min} is the lower size bound, s_{max} is the upper size bound)[4], c is the worst-case approximation of 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` and 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."

For clarity of presentation we denote size annotations with different symbols depending on the base type they annotate. We use n_i to represent a size annotation for an Integer, and r_i for a size annotation on an Image. In our presentation we omit the implicit constraint $s_1 \leq s_2$ present for every size vector $\{s_1, s_2\}$,

3.5.1 Primitive Types

The typing rules for values are given below. The computational cost (c) for a value is always 0.

$$\text{(T-Int)}\quad \frac{}{\Gamma \vdash \texttt{n} : \mathit{Int}^{\{n,n\}}\ \$\ 0, \{n = \texttt{n}\}}$$

A constant integer value has both size annotation variables constrained to the integer's actual value.

[4]This pair can be expanded to a tuple to track more dimensions/aspects, as described in Section 5

$$\text{(T-Image)}\quad \frac{}{\Gamma \vdash \texttt{i} : \mathit{Img}^{\{r,r\}}\ \$\ 0, \{r = \texttt{resolutionof}(\texttt{i})\}}$$

The size of an image is given by its resolution, taken logically to be the number of pixels in the Image, however to simplify our presentation we use only the width of the image. The pair specifies the range of possible resolutions for the image; As with integers, both values will be the same for a specific, concrete instantiation of an Image.

$$\text{(T-Bool)}\quad \frac{}{\Gamma \vdash \texttt{b} : \mathit{Bool}\ \ \$\ 0, \varnothing}$$

A boolean (`true` | `false`) has a negligible fixed size and thus its size annotation is omitted (*i.e.*, {1,1}).

We define some convenience functions to manipulate the type and its size annotation :

$$\texttt{minsize}(t^{\{n_1,n_2\}}) = n_1 \qquad \texttt{maxsize}(t^{\{n_1,n_2\}}) = n_2$$

$$\texttt{base}(t^{\{n_1,n_2\}}) = t$$

For example: $\texttt{minsize}(\mathit{Int}^{\{4,8\}}) = 4 \quad \texttt{base}(\mathit{Int}^{\{4,8\}}) = \mathit{Int} \quad \texttt{maxsize}(\mathit{Img}^{\{2,n\}}) = n$

3.5.2 Subtyping (bounding sizes)

$$\text{(S-Refl)}\quad \tau <: \tau \qquad \text{(S-Trans)}\quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\text{(S-Arrow)}\quad \frac{\tau_2 <: \tau_1 \quad \tau_1' <: \tau_2'}{\tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \qquad \text{(S-Pair)}\quad \frac{\tau_1 <: \tau_2 \quad \tau_1' <: \tau_2'}{\{\tau_1 \times \tau_1'\} <: \{\tau_2 \times \tau_2'\}}$$

$$\text{(S-Sensor)}\quad \frac{\tau_1 <: \tau_2}{\mathit{Sensor}\ \tau_1 <: \mathit{Sensor}\ \tau_2}$$

We define a relation similar to [LH96]'s subtyping relation ($\trianglelefteq$) which allows a weakening of the type to increase a size bound, in order to provide an upper bound of the size of the input relative to work to be completed. In our environment we notice that the correctness of Image processing functions may impacted by the size of the input (*i.e.*, resolution of the image); as such we cannot arbitrarily increase the logical size (resolution) of this data without adverse consequences to functional correctness.

The need to track the lower bound extends into all aspects of the sized typing system, so we use a general sizing/weaken rule (S-Sized) to describe the subtype relationship for specific sized types.

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

Finally, we give the rule for weakening via the subtype relation, which should be clear when considered 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 5 has more on this).

$$\text{(T-Weaken)}\quad \frac{\Gamma \vdash \texttt{e} : \tau_1\ \$\ c, \kappa \quad \tau_1 <: \tau_2}{\Gamma \vdash \texttt{e} : \tau_2\ \$\ c, \kappa \cup \kappa_2}$$

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

3.5.3 Conditional

$$\text{(T-Cond)}\quad \frac{\Gamma \vdash \mathtt{e}_1 : \mathit{Bool}\ \$\ c_1, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau\ \$\ c_2, \kappa_2 \qquad \Gamma \vdash \mathtt{e}_3 : \tau\ \$\ c_3, \kappa_3}{\Gamma \vdash \mathtt{cond\ e_1\ e_2\ e_3} : \tau\ \$\ 1 + c_1 + \mathtt{max}(c_2\ c_3), \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

In the (common) event where terms of the conditional branches are different sizes, the application of weaken can be used to relax the bounds on either side to meet at the lower minimum size and larger maximum size.

In the event that either branch's type contains a size variable, each branch may be weakened to a new, common size variable for the conditional. The example that follows portrays exactly this, in which two images (or expressions of type image) that have different, yet unknown sizes that are supplied as the branches of a conditional only after applying T-Weaken to each to arrive at the new size variables n_5 and n_6:

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

where:

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

3.5.4 Sensors

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

$$\text{(T-ImageSensor)}\quad \frac{}{\Gamma \vdash \mathtt{image} : \mathit{Sensor\ Img}^{\{r_1, r_2\}}\ \$\ 0, \{r_1 \geq \mathtt{r}_{min}\} \cup \{x_2 \leq \mathtt{r}_{max}\}}$$

The `time` sensor returns the number of evaluations (computations) since the beginning of the evaluation.

$$\text{(T-TimeSensor)}\quad \frac{}{\Gamma \vdash \mathtt{time} : \mathit{Sensor\ Int}^{\{n_1, n_2\}}\ \$\ 0, \{n_1 \geq 0\}}$$

$$\text{(T-SensorRead)}\quad \frac{\mathit{type}(\mathtt{get}) = \mathit{Sensor}\ \tau_1 \to \tau_1 \qquad \Gamma \vdash \mathtt{e}_1 : \mathit{Sensor}\ \tau_1\ \$\ c, \kappa_1}{\Gamma \vdash \mathtt{get\ e}_1 : \tau_1\ \$\ c_1 + 1 + \mathit{latentcost}(\mathtt{get}, \mathit{maxsize}(\tau_1)), \kappa}$$

3.5.5 Opcodes

$$\text{(T-Op)}\quad \frac{\mathit{type}(op) = \tau_1 \to \tau_2 \qquad \mathit{constr}(op) = \kappa}{\Gamma \vdash op : \tau_1 \to \tau_2\ \$\ 0, \kappa}$$

$$\text{(T-OpApply)}\quad \frac{\Gamma \vdash op : \tau_1 \to \tau_2 \ \$\ 0, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_1 \ \$\ c, \kappa_2}{\Gamma \vdash \ op\ \ \mathtt{e}_2 : \tau_2 \ \$\ 1 + c + \mathtt{latentcost}(op, \mathtt{maxsize}(\tau_1)), \kappa_1 \cup \kappa_2}$$
$$\text{where } op \in \{op_1 \mid op_2 \mid op_3 \mid \ldots\}$$

The `latentcost()` function [RG94] 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 a function of the total number of pixels in the image.

3.5.6 Let binding

An instance of a variable has whatever type is assumed for it in the typing environment Γ. A variable has no computational cost or constraints associated with it, rather the costs and constraints are assigned when variables are bound (*i.e.*, in the `let` term).

$$\text{(T-Var)}\quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \ \$\ 0, \varnothing}$$

In a `let` term, we take the sum of the costs of the let-bound expressions as well as the union of all of their associated constraints.

$$\text{(T-LetN)}\quad \frac{\Gamma \vdash \mathtt{e}_1 : \tau_1 \ \$\ c_1, \kappa_1 \ \ldots\ \Gamma \vdash \mathtt{e}_n : \tau_n \ \$\ c_n, \kappa_n \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \mathtt{e}_{n+1} : \tau_{n+1} \ \$\ c_{n+1}, \kappa_{n+1}}{\Gamma \vdash \mathtt{let}\ \{x_1 = \mathtt{e}_1, \ldots, x_n = \mathtt{e}_n\}\ \mathtt{in}\ \mathtt{e}_{n+1} : \tau_{n+1} \ \$\ c_{n+1} + \sum_{i=1}^{n} c_i, \kappa_{n+1} \cup \bigcup_{i=1}^{n} \kappa_i}$$

The cost defined in this rule is an over-estimate of the total cost as the let-bound symbols (some $\mathtt{x}_i$) may not occur in the evaluation path of $\mathtt{e}_{n+1}$. These costs will not, however, be "charged" twice, as a variable itself has cost zero when computing the cost of $\mathtt{e}_{n+1}$.

3.5.7 Triggers

A trigger's cost and constraint takes much the same form as the `cond` and `let` instances from which the trigger is derived.

$$\text{(T-Trig)}\frac{\Gamma \vdash \mathtt{e}_i : \tau_i \ \$c_i, \kappa_i \ (\text{for } i \in 1..n) \quad \Gamma, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+1} : \mathit{Bool} \ \$c_{n+1}, \kappa_{n+1} \quad \Gamma, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+2} : \tau \ \$c_{n+2}, \kappa_{n+2}}{\Gamma \vdash \mathtt{trigger}\ \{\mathtt{x}_i = \mathtt{e}_i\}^{1..n} \mathtt{e}_{n+1}\ \mathtt{e}_{n+2} : \tau \ \$\ \mathcal{T} * (c_{n+1} + \sum_{i=1}^{n} c_i) + c_{n+2}, \kappa_{n+1} \cup \kappa_{n+2} \cup \bigcup_{i=1}^{n} \kappa_i}$$
$$\text{where } \mathcal{T} \text{ is a new cost variable from the set of unused cost variables, } \mathcal{C}$$
Specific values for $\mathcal{T}$ may be provided by explicit user bounds or other means.

3.5.8 Typing of the Let-store, ν

We can assign a type (more accurately sequence of types) to a variable store ν if we have assumed types for all of the variables contained within the store. Similarly we associate a cost and constraint set with the store (for use in interim steps of typing derivations).

$$\text{(T-Nu)}\quad \frac{\Gamma, \Gamma' \vdash \nu(x_i) : \Gamma'(x_i) \ \$\ c_i, \kappa_i \qquad (\text{for every } x_i \in \{x_1, \ldots, x_n\} = \mathit{Domain}(\nu))}{\Gamma \vdash \nu : \Gamma' \ \$\ \sum_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \kappa_i}$$

Finally we can combine an expression *and* a typed variable store into a typed pair by discharging the typing assumptions of the variable store, as shown below.

$$\text{(T-Complete)}\quad \frac{\Gamma \vdash \mathsf{e} : \tau \ \$ \ c_1, \kappa_1 \qquad \varnothing \vdash \nu : \Gamma \ \$ c_2, \kappa_2}{\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c_1 + c_2, \kappa_1 \cup \kappa_2}$$

3.6 Soundness of Core STEP

This Section proves *Soundness* for the Core STEP (*Soundness* = *Progress*+*Preservation*). *Progress* means that every expression is either a value or can take an evaluative step (*i.e.*, expressions don't get stuck), while *Preservation* means that every typed expression that takes an evaluative step results in another typed expression (*i.e.*, evaluation is type preserving). We begin by proving some Lemmas that will be useful for our proofs of Progress and Preservation.

3.6.1 Lemma: L-Var

Suppose $\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathsf{e} \mid \nu$ is closed.
If $\mathsf{e} = x_i$, then for some c' and κ':
$\varnothing \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa'$

Proof: By the structure of the derivation of $\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c, \kappa$ where $\mathsf{e} = x_i$.

$$\frac{\dfrac{\Gamma(x_i) = \tau}{\Gamma \vdash x_i : \tau \ \$ \ 0, \varnothing}(\text{T-Var}) \quad \dfrac{\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \quad (\text{for every } x_j \in \{x_1, \ldots, x_n\} = \mathit{Domain}(\nu))}{\varnothing \vdash \nu : \Gamma \ \$ \ c, \kappa}(\text{T-Nu})}{\varnothing \vdash x_i \mid \nu : \Gamma(x_i) \ \$ \ c, \kappa}(\text{T-Complete})$$

From the premise of T-Nu: $\Gamma \vdash \nu(x_i) : \Gamma(x_i) \ \$ \ c_i, \kappa_i$
From the premise of T-Var: $\Gamma(x_i) = \tau$
Combining these we have: $\Gamma \vdash \nu(x_i) : \tau \ \$ \ c_i, \kappa_i$

$$\frac{\Gamma \vdash \nu(x_i) : \tau \ \$ \ c_i, \kappa_i \qquad \varnothing \vdash \nu : \Gamma \ \$ \ c, \kappa}{\varnothing \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa'}(\text{T-Complete})$$

3.6.2 Lemma(s): Inversion+ν

Suppose $\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c, \kappa$ is the last judgment $\mathcal{J}_0$ in a typing derivation tree $\mathcal{D}$. Then:
(1) The left premise of this judgment in $\mathcal{D}$ is $\Gamma \vdash \mathsf{e} : \tau \$ c''', \kappa'''$ (say $\mathcal{J}_1$) for some c''' and κ'''
(2) If $\mathcal{J}_2$ is a premise of $\mathcal{J}_1$ of the form $\Gamma' \vdash \mathsf{e}' : \tau' \$ c', \kappa'$ for some c', κ' then $\Gamma' \vdash \mathsf{e}' \mid \nu : \tau' \$ c', \kappa'$ can be derived for all such $\mathcal{J}_2$.

Proof: (Generic) By the structure of the derivation of $\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c, \kappa$ (below). In the general form, to have reached $\varnothing \vdash \mathsf{e} \mid \nu : \tau \ \$ \ c, \kappa$, we must have an application of T-Complete with right-side premise $\varnothing \vdash \nu : \Gamma \ \$ \ c'', \kappa''$ and on the left side of the derivation we find the individual sub-premises to type the expression e. We call this rule T-*Rule* as a placeholder for specific instances of e and similarly denote the sub-premises as $\mathcal{J}_2$ and the consequent (in which e is given a type) as $\mathcal{J}_1$. We can apply

T-Complete with the right hand side premise of the existing T-Complete to each sub-premise of $\mathcal{J}_2$ individually to arrive at our conclusion. We detail the specific individual cases below.

$$\dfrac{\dfrac{\mathcal{J}_2}{\mathcal{J}_1 = \Gamma \vdash \mathtt{e} : \tau\$c''', \kappa'''}(\text{T-}\textit{Rule}) \quad \dfrac{\Gamma \vdash \nu(x_j) : \Gamma(x_j)\ \$\ c_j, \kappa_j \text{ (for every } x_j \in \{x_1, \ldots, x_n\} = \texttt{\textit{Domain}}(\nu))}{\varnothing \vdash \nu : \Gamma\ \$\ c'', \kappa''}(\text{T-Nu})}{\mathcal{J}_0 = \varnothing \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa}(\text{T-Complete})$$

Case L-Cond:

Suppose $\Gamma \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$ and $\mathtt{e} \mid \nu$ is closed.

If $\mathtt{e} = \mathtt{cond\ e_1\ e_2\ e_3}$ then for some c_1', c_2', c_3' and $\kappa_1', \kappa_2', \kappa_3'$:

$\varnothing \vdash \mathtt{e}_1 \mid \nu : \mathit{Bool}\ \$\ c_1', \kappa_1'$

$\varnothing \vdash \mathtt{e}_2 \mid \nu : \tau\ \$\ c_2', \kappa_2'$

$\varnothing \vdash \mathtt{e}_3 \mid \nu : \tau\ \$\ c_3', \kappa_3'$

Proof: Using the generic proof above, where:

$\mathcal{J}_2 = \Gamma \vdash \mathtt{e}_1 : \mathit{Bool}\ \$\ c_1, \kappa_1 \quad \Gamma \vdash \mathtt{e}_2 : \tau\ \$\ c_2, \kappa_2 \quad \Gamma \vdash \mathtt{e}_3 : \tau\ \$\ c_3, \kappa_3$

$\mathcal{J}_1 = \Gamma \vdash \mathtt{cond\ e_1\ e_2\ e_3} : \tau\ \$\ (1 + c_1 + (max)(c_2, c_3))), (\kappa_1 \cup \kappa_2 \cup \kappa_3)$

T-*Rule* = T-Cond.

Case L-Get:

Suppose $\Gamma \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$ and $\mathtt{e} \mid \nu$ is closed.

If $\mathtt{e} = \mathtt{get\ e_1}$ then for some c_1' and κ_1':

$\varnothing \vdash \mathtt{e}_1 \mid \nu : \mathit{Sensor}\ \tau_1\ \$\ c_1', \kappa_1'$

Proof: Using the generic proof above, where:

$\mathcal{J}_2 = \Gamma \vdash \mathtt{e}_1 : \mathit{Sensor}\ \tau_1\ \$\ c_1, \kappa_1$

$\mathcal{J}_1 = \Gamma \vdash \mathtt{get\ e_1} : \tau_1\ \$\ c_1'', \kappa_1''$

T-*Rule* = T-SensorRead.

Case L-OpApply:

Suppose $\Gamma \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$ and $\mathtt{e} \mid \nu$ is closed.

If $\mathtt{e} = op\ \mathtt{e_2}$ then for some c_1', c_2' and $\kappa_1', kappa_2'$:

$\varnothing \vdash op \mid \nu : \tau_1 \to \tau_2\$\ c_1', \kappa_1'$

$\varnothing \vdash \mathtt{e}_2 \mid \nu : \tau_1\ \$\ c_2', \kappa_2'$

Proof: Using the generic proof above, where:

$\mathcal{J}_2 = \Gamma \vdash op : \tau_1 \to \tau_2\$\ 0, \kappa_1 \quad \Gamma \vdash \mathtt{e}_2 : \tau_1\ \$\ c, \kappa_2$

$\mathcal{J}_1 = \Gamma \vdash op\ \mathtt{e}_2 : \tau\$c, \kappa_1 \cup \kappa_2$

T-*Rule* = T-OpApply.

Case L-LetN:

Suppose $\Gamma \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$ and $\mathtt{e} \mid \nu$ is closed.

If $\mathtt{e} = \mathtt{let}\ \{x_1 = \mathtt{e}_1, \ldots, x_n = \mathtt{e}_n\}\ \mathtt{in}\ \mathtt{e}_{n+1}$ then for some $c_1' \ldots c_{n+1}'$ and $\kappa_1' \ldots \kappa_{n+1}'$:

$\varnothing \vdash \mathtt{e}_1 \mid \nu : \tau_1\ \$\ c_1', \kappa_1'$

$\vdots$

$\varnothing \vdash \mathtt{e}_n \mid \nu : \tau_n\ \$\ c_n', \kappa_n'$

$\varnothing, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \mathtt{e}_{n+1} \mid \nu : \tau_{n+1}\ \$\ c_{n+1}', \kappa_{n+1}'$

Proof: Using the generic proof above, where:

$\mathcal{J}_2 = \Gamma \vdash \mathtt{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \ldots \ \Gamma \vdash \mathtt{e}_n : \tau_n \ \$ \ c_n, \kappa_n \quad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \mathtt{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1}, \kappa_{n+1}$

$\mathcal{J}_1 = \Gamma \vdash \mathtt{let} \ \{x_1 = \mathtt{e}_1, \ldots, x_n = \mathtt{e}_n\} \ \mathtt{in} \ \mathtt{e}_{n+1} : \tau \ \$ \ c_{n+1} + \sum_{i=1}^{n} c_i, \kappa_{n+1} \cup \bigcup_{i=1}^{n} \kappa_i$

T-*Rule* = T-LetN.

Case L-Trigger:

Suppose $\Gamma \vdash \mathtt{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathtt{e} \mid \nu$ is closed.

If $\mathtt{e} = \mathtt{trigger} \ \{\mathtt{x}_i = \mathtt{e}_i\}^{i \in 1..n} \mathtt{e}_{n+1} \ \mathtt{e}_{n+2}$ then for some $c'_1 \ldots c'_{n+2}$ and $\kappa'_1 \ldots \kappa'_{n+2}$:

$\varnothing \vdash \mathtt{e}_1 \mid \nu : \tau_1 \ \$ \ c'_1, \kappa'_1$

$\ldots$

$\varnothing \vdash \mathtt{e}_n \mid \nu : \tau_n \ \$ \ c'_n, \kappa'_n$

$\varnothing, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+1} \mid \nu : \mathit{Bool} \ \$ \ c'_{n+1}, \kappa'_{n+1}$

$\varnothing, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+2} \mid \nu : \tau \ \$ \ c'_{n+2}, \kappa'_{n+2}$

Proof: Using the generic proof above, where:

$\mathcal{J}_2 = \Gamma \vdash \mathtt{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \ldots \ \Gamma \vdash \mathtt{e}_n : \tau_n \ \$ \ c_n, \kappa_n \quad \Gamma, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+1} : \mathit{Bool} \ \$ \ c_{n+1}, \kappa_{n+1} \quad \Gamma, \{\mathtt{x}_i : \tau_i\}^{1..n} \vdash \mathtt{e}_{n+2} : \tau \ \$ \ c_{n+2}, \kappa_{n+2}$

$\mathcal{J}_1 = \Gamma \vdash \mathtt{trigger} \ \{\mathtt{x}_i = \mathtt{e}_i\}^{i \in 1..n} \mathtt{e}_{n+1} \ \mathtt{e}_{n+2} : \tau \ \$ \ \mathcal{T} * (c_{n+1} + \sum_{i=1}^{n} c_i) + c_{n+2}, \kappa_{n+1} \cup \kappa_{n+2} \cup \bigcup_{i=1}^{n} \kappa_i$

T-*Rule* = T-Trig.

3.6.3 Theorem [Progress]

Suppose $\mathtt{e} \mid \nu$ is closed and $\varnothing \vdash \mathtt{e} \mid \nu : \tau \ \$ \ c, \kappa$ (for some cost c and constraint set κ)

Either: (1) $\mathtt{e}$ is a value or (2) there exists some $\mathtt{e}'$ and store ν' such that for every t:

$(\mathtt{e} \mid \nu \mid t) \ \rightarrow \ (\mathtt{e}' \mid \nu' \mid t + 1)$.

Proof: By induction on the number of *unique* sub-derivations of a typing derivation of $\varnothing \vdash \mathtt{e} \mid \nu : \tau \ \$ \ c, \kappa$.

We proceed by analysis of the shape of $\mathtt{e}$ to show that the property holds for the larger derivation.

Case $\mathtt{e} = \mathtt{v}$:

Satisfied trivially as these terms are values.

Case $\mathtt{e} = \mathtt{cond} \ \mathtt{e}_1 \ \mathtt{e}_2 \ \mathtt{e}_3$:

Applying the Induction Hypothesis to a derivation whose final judgment is $\Gamma \vdash \mathtt{e}_1 \mid \nu : \mathit{Bool} \ \$ \ c_1, \kappa_1$ (*v*ia L-Cond) we can verify that progress holds for $\mathtt{e}_1 \mid \nu$ (*i.e.*, $\mathtt{e}_1$ is a value or $\mathtt{e}_1 \mid \nu$ can take an evaluative step). So either (1) $\mathtt{e}_1$ is a value (specifically, a Boolean) and E-IfTrue or E-IfFalse can be applied or $\mathtt{e}_1 \mid \nu$ can take an evaluative step such that E-If applies and $(\mathtt{e} \mid \nu \mid t) \rightarrow (\mathtt{cond} \ \mathtt{e}'_1 \ \mathtt{e}_2 \ \mathtt{e}_3 \mid \nu' \mid t + 1)$.

Case $\mathtt{e} = x_i$ with $x_i : \tau_i$ and $x_i \in \mathit{Domain}(\nu)$ (as $\mathtt{e} \mid \nu$ is closed):

By the definition of ν (and closure of the pair $\mathtt{e} \mid \nu$) we re-write $\varnothing \vdash x_i \mid \nu : \tau$ as $\varnothing \vdash x_i \mid \{x_i \mapsto \mathtt{e}_i\} \cup \nu_+ : \tau$. By the definition of ν either $\mathtt{e}_i = \mathtt{v}_i$ and E-Var applies without premise or $\mathtt{e}_i$ is a composite expression. To apply E-Var1 we require progress for $\mathtt{e}_i \mid \nu$. By the

definition of ν we can re-write $\mathtt{e}_i$ as $\nu(x_i)$ for which progress holds via L-VAR (to which we apply the the Inductive Hypothesis). Hence E-VAR or E-VAR1 may be applied and progress holds for this term.

Case $\mathtt{e} = \mathtt{let}\ \{x_1 = \mathtt{e}_1, \ldots, x_n = \mathtt{e}_n\}\ \mathtt{in}\ \mathtt{e}_{n+1}$:
E-LETN takes an evaluative step without premise such that $(\mathtt{e} \mid \nu \mid t) \rightarrow (\mathtt{e}_{n+1} \mid \nu \cup \{x_1 \mapsto \mathtt{e}_1, \ldots, x_n \mapsto \mathtt{e}_n\} \mid t+1)$.

Case $\mathtt{e} = \mathtt{trigger}\ \{x_1 = \mathtt{e}_1, \ldots, x_n = \mathtt{e}_n\}\ \mathtt{e}_{n+1}\ \mathtt{e}_{n+2}$:
E-TRIGGERLET applies without premise for a single step of evaluation.
In the event that the let assignment set is empty (*i.e.*, $\mathtt{e} = \mathtt{trigger}\ \mathtt{e}_{n+1}\ \mathtt{e}_{n+2}$) then E-TRIGGER1 applies without premise.

Case $\mathtt{e} = \mathtt{get}\ \mathtt{e}_1$:
Either $\mathtt{e}_1$ is a value and E-GETAPPLY applies (or E-GETTIME if $\mathtt{e}_1 = \mathtt{time}$), or by L-GET and the Inductive Hypothesis, we show progress holds on $(\mathtt{e}_1 \mid \nu)$ so $\mathtt{e}_1 \mid \nu \mid t \rightarrow \mathtt{e}_1' \mid \nu' \mid t+1$ and E-GET1 applies such that $(\mathtt{e} \mid \nu \mid t) \rightarrow (\mathtt{get}\ \mathtt{e}_1' \mid \nu' \mid t+1)$

Case $\mathtt{e} = op\ \mathtt{e}_2$:
Either $\mathtt{e}_2$ is a value and E-OPAPPLY applies, or by L-OPAPPLY and the Inductive Hypothesis, progress holds on $\mathtt{e}_2 \mid \nu$ so $\mathtt{e}_2 \mid \nu \mid t \rightarrow \mathtt{e}_2' \mid \nu' \mid t+1$ and E-OP1 applies such that $(\mathtt{e} \mid \nu \mid t) \rightarrow (\mathrm{op}\ \mathtt{e}_2' \mid \nu' \mid t+1)$

3.6.4 Theorem [Preservation]

Suppose $\mathtt{e} \mid \nu$ is closed and $\varnothing \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$.
If $(\mathtt{e} \mid \nu \mid t) \rightarrow (\mathtt{e}' \mid \nu' \mid t+1)$, then for some cost c' and constraint set κ':
$\varnothing \vdash \mathtt{e}' \mid \nu' : \tau\ \$\ c', \kappa'$

Proof: By induction on the number of *distinct* sub-derivations of a typing derivation of $\varnothing \vdash \mathtt{e} \mid \nu : \tau\ \$\ c, \kappa$.
We proceed by analysis of the shape of $\mathtt{e}$ to show that the property holds for the larger derivation.

Case $\mathtt{e} = \mathtt{v}$:
These cases are satisfied trivially, as these terms are values and do not make a further evaluative step.

Case $\mathtt{e} = \mathtt{cond}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3$:
We consider each possible evaluative case, individually.

Case E-IFTRUE:
$(\mathtt{cond}\ \mathtt{true}\ \mathtt{e}_2\ \mathtt{e}_3 \mid \nu \mid t) \rightarrow (\mathtt{e}_2 \mid \nu \mid t+1)$
We consider the typing derivation of $\mathtt{e} \mid \nu$ in which we have $\Gamma \vdash \mathtt{e}_2 : \tau\ \$\ c_2, \kappa_2$ (from the premise T-COND) and $\nu : \Gamma\ \$\ c'', \kappa''$ (from the premise T-COMPLETE). We combine them in a new application of T-COMPLETE to verify $\varnothing \vdash \mathtt{e}_2 \mid \nu : \tau\ \$\ c', \kappa'$.

Case E-IFFALSE:

$(\texttt{cond false } \mathtt{e}_2\ \mathtt{e}_3 \mid \nu \mid t) \rightarrow (\mathtt{e}_3 \mid \nu \mid t+1)$

We consider the typing derivation of $\mathtt{e} \mid \nu$ in which we have $\Gamma \vdash \mathtt{e}_3 : \tau \ \$ \ c_3, \kappa_3$ (from the premise T-Cond) and $\nu : \Gamma \ \$ \ c'', \kappa''$ (from the premise T-Complete). We combine them in a new application of T-Complete to verify $\varnothing \vdash \mathtt{e}_3 \mid \nu : \tau \ \$ \ c', \kappa'$.

Case E-If:

$(\texttt{cond } \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3 \mid \nu \mid t) \rightarrow (\texttt{cond } \mathtt{e}'_1\ \mathtt{e}_2\ \mathtt{e}_3 \mid \nu' \mid t+1)$

By the premise of T-Cond $\mathtt{e}_1 : \mathit{Bool}$ and by L-Cond $\mathtt{e}_1 \mid \nu : \mathit{Bool}$. By applying the Inductive Hypothesis to a judgment with this as its last term, we obtain $\mathtt{e}'_1 \mid \nu' : \mathit{Bool}$.

Looking at the typing derivation of $\mathtt{e}'_1 \mid \nu' : \mathit{Bool}$:

$$\frac{g = (\Gamma \vdash \mathtt{e}'_1 : \mathit{Bool} \ \$ \ c_1, \kappa_1) \qquad \dfrac{\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \text{ (for every } x_j \in \{x_1, \ldots, x_n\} = \mathit{Domain}(\nu))}{h = (\varnothing \vdash \nu' : \Gamma \ \$ \ c'', \kappa'')}\text{(T-Nu)}}{\varnothing \vdash \mathtt{e}'_1 \mid \nu' : \mathit{Bool} \ \$ \ c'_1, \kappa'_1}\text{(T-Complete)}$$

Now we apply T-Cond to g and the terms from the derivation of $\texttt{cond } \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3 \mid \nu$ (if needed, see α in L-Cond for clarity):

$$\frac{\Gamma \vdash \mathtt{e}'_1 : \mathit{Bool} \ \$ \ c_1, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau \ \$ \ c_2, \kappa_2 \qquad \Gamma \vdash \mathtt{e}_3 : \tau \ \$ \ c_3, \kappa_3}{i = (\Gamma \vdash \texttt{cond } \mathtt{e}'_1\ \mathtt{e}_2\ \mathtt{e}_3 : \tau \ \$ \ c'_4, \kappa'_4)}\text{(T-Cond)}$$

Now we apply T-Complete to h and i to verify: $\varnothing \vdash \texttt{cond } \mathtt{e}'_1\ \mathtt{e}_2\ \mathtt{e}_3 \mid \nu' : \tau \ \$ \ c', \kappa'$

Case $\mathtt{e} = x_i$ with $x_i : \tau_i$ and $x_i \in \mathit{Domain}(\nu)$ (as $\mathtt{e} \mid \nu$ is closed):

By the definition of ν (and closure of the pair $\mathtt{e} \mid \nu$) we rewrite $\vdash x_i \mid \nu : \tau \ \$ \ c', \kappa'$ as $\vdash x_i \mid \{x_i \mapsto \nu(x_i)\} \cup \nu_+ : \tau \ \$ \ c', \kappa'$. We now consider the possible evaluation cases individually:

Case E-Var1 ($\nu(x_i) = \mathtt{v}_i$) :

$(x_i \mid \{x_i \mapsto \mathtt{v}_i\} \cup \nu_+ \mid t) \rightarrow (\mathtt{v}_i \mid \{x_i \mapsto \mathtt{v}_i\} \cup \nu_+ \mid t+1)$

We know from the derivation of $\mathtt{e} \mid \nu$, the premise of T-Nu gives $\Gamma \vdash \nu(x_i) : \tau_i \ \$ \ c_i, \kappa_i$. As $\nu(x_i) = \mathtt{v}_i$ and $\Gamma(x_i) = \tau_i = \tau$ we can re-write this as $g = (\Gamma \vdash \mathtt{v}_i : \tau \ \$ \ c_i, \kappa_i)$.

Again from the derivation we also have the premise of T-Complete: $h = (\varnothing \vdash \nu : \Gamma \ \$ \ c'', \kappa'')$. Combining g and h under T-Complete we can verify $\vdash \mathtt{v}_i \mid \nu : \tau \ \$ \ c', \kappa' \ = \vdash \mathtt{e}' \mid \nu : \tau \ \$ \ c', \kappa'$.

Case E-Var2 ($\nu(x_i) = \mathtt{e}_i$):

$(x_i \mid \{x_i \mapsto \mathtt{e}_i\} \cup \nu_+ \mid t) \rightarrow (x_i \mid \{x_i \mapsto \mathtt{e}'_i\} \cup \nu'_+ \mid t+1)$

L-Var gives us $(\varnothing \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa')$, which we rewrite by the definition of ν as $(\varnothing \vdash \mathtt{e}_i \mid \nu : \tau \ \$ \ c', \kappa')$. Applying the Inductive Hypothesis we have that $(\varnothing \vdash \mathtt{e}'_i \mid \nu' : \tau \ \$ \ c''', \kappa''')$. From the typing derivation of this expression we have a right side premise for T-Complete: $h = (\varnothing \vdash \nu' : \Gamma \ \$ \ c'', \kappa'')$. Combining h with our prior left hand premise of T-Complete, the T-Var's $\Gamma \vdash x_i : \tau \ \$ \ 0, \varnothing$, we have:

$\vdash x_i \mid \nu' : \tau \ \$ \ c', \kappa' \ = \vdash \mathtt{e}' \mid \nu' : \tau \ \$ \ c', \kappa'$.

Case $\texttt{e} = \texttt{let}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{in}\ \texttt{e}_{n+1}$:

Case E-LetN:

$(\texttt{let}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{in}\ \texttt{e}_{n+1} \mid \nu \mid t) \rightarrow (\texttt{e}_{n+1} \mid \nu \cup \{x_1 \mapsto \texttt{e}_1, \ldots, x_n \mapsto \texttt{e}_n\} \mid t+1)$

Thus our goal is to type: $\texttt{e}_{n+1} \mid \nu \cup \{x_1 \mapsto \texttt{e}_1, \ldots, x_n \mapsto \texttt{e}_n\}$

We first refer to the typing derivation of $\texttt{e} \mid \nu$ (and define aliases to save space):

$g = (\Gamma \vdash \texttt{e}_1 : \tau_1\ \$\ c_1, \kappa_1\ \ldots\ \Gamma \vdash \texttt{e}_n : \tau_n\ \$\ c_n, \kappa_n)$

$h = (\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{e}_{n+1} : \tau_{n+1}\ \$\ c_{n+1}, \kappa_{n+1})$

$$\mathcal{Y} = \frac{g \qquad h}{\Gamma \vdash \texttt{let}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{in}\ \texttt{e}_{n+1} : \tau_{n+1}\ \$\ c_{n+1} + \sum_{i=1}^{n} c_i, \kappa_{n+1} \cup \bigcup_{i=1}^{n} \kappa_i}(\text{T-LetN})$$

$$\frac{\mathcal{Y} \qquad \dfrac{i = (\Gamma \vdash \nu(x_j) : \Gamma(x_j)\ \$\ c_j, \kappa_j \quad \text{for every } x_j \in \{x_1, \ldots, x_n\} = \textit{Domain}(\nu))}{\varnothing \vdash \nu : \Gamma\ \$\ c'', \kappa''}(\text{T-Nu})}{\varnothing \vdash \texttt{e} \mid \nu : \tau \$\ c_1, \kappa_1}(\text{T-Complete})$$

We call $\nu_1 = \{x_1 \mapsto \texttt{e}_1, \ldots, x_n \mapsto \texttt{e}_n\}$ and call to g and h show that each $\nu_1(x_i) : \Gamma(x_i)$ for $x_i \in \textit{Domain}(\nu_1)$. We see $\nu' = \nu \cup \nu_1$ and now we have what we need to apply T-Complete and show our desired result.

$$\frac{\text{h} \qquad \dfrac{\Gamma \vdash \nu'(x_j) : \Gamma(x_j)\ \$\ c_j, \kappa_j \quad \text{for every } x_j \in \{x_1, \ldots, x_n\} = \textit{Domain}(\nu'))}{\varnothing \vdash \nu' : \Gamma\ \$\ c'', \kappa''}(\text{T-Nu})}{\varnothing \vdash \texttt{e}_{n+1} \mid \nu' : \tau \$\ c', \kappa'}(\text{T-Complete})$$

Case $\texttt{e} = \texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2$:

Case E-Trigger1:

$(\texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2 \mid \nu \mid t) \rightarrow (\texttt{cond}\ \texttt{e}_1\ \ \texttt{e}_2\ (\texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2) \mid \nu \mid t+1)$

From the typing derivation of $\texttt{e} \mid \nu$ we have $\texttt{e}_1 : \textit{Bool}$, $\texttt{e}_2 : \tau$, $\texttt{e} : \tau$, and $\varnothing \vdash \nu : \Gamma\ \$\ c'', \kappa''$. We combine the first three under T-Cond and the result with the last under T-Complete to obtain our desired result.

Case E-TriggerDeg:

$(\texttt{trigger}\ \texttt{v}_1\ \texttt{e}_2 \mid \nu \mid t) \rightarrow (\texttt{cond}\ \texttt{v}_1\ \ \texttt{e}_2\ (\texttt{trigger}\ \texttt{e}_1\ \texttt{e}_2) \mid \nu \mid t+1)$

Identical to the above.

Case $\texttt{e} = \texttt{trigger}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2}$:

Case E-TriggerLet:

$(\texttt{trigger}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2} \mid \nu \mid t)$

$\rightarrow (\texttt{let}\ \{x_1 = \texttt{e}_1, \ldots, x_n = \texttt{e}_n\}\ \texttt{in}\ (\texttt{cond}\ \texttt{e}_{n+1}\ \texttt{e}_{n+2}\ \texttt{e}) \mid \nu \mid t+1)$

From the typing derivation of $\texttt{e} \mid \nu$ we have $\texttt{e}_{n+1} : \textit{Bool}$, $\texttt{e}_{n+2} : \tau$, and $\texttt{e} : \tau$. We combine these under T-Cond, the result with the variable premises under T-LetN and finally this with $\varnothing \vdash \nu : \Gamma\ \$\ c'', \kappa''$ under T-Complete to obtain our desired result.

Case $\mathtt{e} = op\ \mathtt{e}_1$:

Case E-OpApply:

$(op\ \mathtt{v}_1 \mid \nu \mid t) \rightarrow (\mathtt{v}_2 \mid \nu \mid t+1)$ The function ***Apply*** by definition, returns a value (which is inherently typed). By the typing derivation of $op\ \mathtt{v}_1 \mid \nu$ we find $\varnothing \vdash \nu : \Gamma\ \$\ c', \kappa'$ and can apply T-Complete to obtain our desired result.

Case E-Op1:

$(op\ \mathtt{e}_1 \mid \nu \mid t) \rightarrow (op\ \mathtt{e}_1' \mid \nu' \mid t+1)$

By the premise of T-OpApply $\mathtt{e}_1 : \tau_1$ and by L-OpApply $\mathtt{e}_1 \mid \nu : \tau_1$. By the Inductive Hypothesis we have $\mathtt{e}_1' \mid \nu' : \tau_1$.

Looking at the typing derivation of $\mathtt{e}_1' \mid \nu' : \tau_1$:

$$\frac{g = (\Gamma \vdash \mathtt{e}_1' : \tau_1 \$\ c_1, \kappa_1) \qquad h = (\varnothing \vdash \nu' : \Gamma\ \$\ c', \kappa')}{\varnothing \vdash \mathtt{e}_1' \mid \nu' : \tau_1 \$\ c', \kappa'}(\text{T-Complete})$$

Now we apply T-OpApply to g and the terms from the derivation of $op\ \mathtt{e}_1 \mid \nu$ (if needed, see α in L-OpApply for clarity):

$$\frac{\Gamma \vdash op : \tau_1 \rightarrow \tau_2 \$\ c_1, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_1' : \tau_1\ \$\ c_2, \kappa_2}{i = (\Gamma \vdash\ op\ \mathtt{e}_1'\ : \tau_2\ \$\ c', \kappa')}(\text{T-OpApply})$$

Now we apply T-Complete to h and i to verify: $\varnothing \vdash\ op\ \mathtt{e}_1' \mid \nu' : \tau\ \$\ c''', \kappa'''$

Case $\mathtt{e} = \mathtt{get}\ \mathtt{e}_1$:

Case E-GetApply:

$(\mathtt{get}\ \mathtt{v}_1 \mid \nu \mid t) \rightarrow (\mathtt{v}_2 \mid \nu \mid t+1)$

The function `get` by definition, returns a value (which is inherently typed). By the typing derivation of $\mathtt{get}\ \mathtt{v}_1 \mid \nu$ we find $\varnothing \vdash \nu : \Gamma\ \$\ c', \kappa'$ and can apply T-Complete to obtain our desired result.

Case E-Get1:

$(\mathtt{get}\ \mathtt{e}_1 \mid \nu \mid t) \rightarrow (\mathtt{get}\ \mathtt{e}_1' \mid \nu' \mid t+1)$

By the premise of T-SensorRead $\mathtt{e}_1 : \mathit{Sensor}\ \tau_1$ and by L-Get $\mathtt{e}_1 \mid \nu : \mathit{Sensor}\ \tau_1$. By the Inductive Hypothesis we have $\mathtt{e}_1' \mid \nu' : \mathit{Sensor}\ \tau_1$.

Looking at the typing derivation of $\mathtt{e}_1' \mid \nu' : \mathit{Sensor}\ \tau_1$:

$$\frac{g = (\Gamma \vdash \mathtt{e}_1' : \mathit{Sensor}\ \tau_1 \$\ c_1, \kappa_1) \qquad h = (\varnothing \vdash \nu' : \Gamma\ \$\ c', \kappa')}{\varnothing \vdash \mathtt{e}_1' \mid \nu' : \mathit{Sensor}\ \tau_1 \$\ c', \kappa'}(\text{T-Complete})$$

Now we apply T-SensorRead to g and the terms from the derivation of $\mathtt{get}\ \mathtt{e}_1 \mid \nu$ (if needed, see α in L-Get for clarity):

$$\frac{\Gamma \vdash \mathtt{get} : \mathit{Sensor}\ \tau_1 \rightarrow \tau_1 \$\ c_1, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_1' : \mathit{Sensor}\ \tau_1\ \$\ c_2, \kappa_2}{i = (\Gamma \vdash\ \mathtt{get}\ \mathtt{e}_1'\ : \tau_1\ \$\ c', \kappa')}(\text{T-SensorRead})$$

Now we apply T-Complete to h and i to verify: $\varnothing \vdash\ \mathtt{get}\ \mathtt{e}_1' \mid \nu' : \tau_1\ \$\ c''', \kappa'''$

4 Applications of the formalism

In this section we present concrete instantiations of the formalism presented in the previous section, in order to illustrate its benefit.

4.1 Additional Syntax

In addition to defining specific primitive operators (Opcodes) we also define the notion of a pair to allow these operators to accept multiple inputs.

e	::=		***expressions***
		{e,e}	*pair*
		ft	*flowtypes*
v	::=		***values***
		{v,v}	*pair*
op	::=		***opcodes***
		`fst`	*first projection of a pair*
		`snd`	*second projection of a pair*
		`sizedimage`	*allocate an image sensor (supporting specific sizes)*
		`add`	*add integers*
		`facect`	*count faces in an image*
		`resample`	*change the resolution of an image*
ft	::=		***flowtypes***
		`deadline n` e	*define a timing constraint*
τ	::=		***types***
		$\tau \times \tau$	*pair*

4.2 Typing and Evaluation Rules for Pairs

Evaluation (strict) and typing rules for pairs are handled in a standard way.

$$\text{(T-PairCons)}\quad \frac{\Gamma \vdash \mathtt{e}_1 : \tau_1 \ \$\ c_1, \kappa_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2 \ \$\ c_2, \kappa_2}{\Gamma \vdash \{\mathtt{e}_1, \ \mathtt{e}_2\} : \tau_1 \times \tau_2 \ \$\ 1 + c_1 + c_2, \kappa_1 \cup \kappa_2}$$

$$\text{(T-PairFirst)}\quad \frac{\Gamma \vdash \mathtt{fst} : \tau_1 \times \tau_2 \to \tau_1 \ \$\ 0, \varnothing \qquad \Gamma \vdash \mathtt{e} : \tau_1 \times \tau_2 \ \$\ c, \kappa}{\Gamma \vdash \mathtt{fst\ e} : \tau_1 \ \$\ 1 + c, \kappa}$$

$$\text{(T-PairSecond)}\quad \frac{\Gamma \vdash \mathtt{snd} : \tau_1 \times \tau_2 \to \tau_2 \ \$\ 0, \varnothing \qquad \Gamma \vdash \mathtt{e} : \tau_1 \times \tau_2 \ \$\ c, \kappa}{\Gamma \vdash \mathtt{snd\ e} : \tau_2 \ \$\ 1 + c, \kappa}$$

$$\text{(E-PairFst)}\ \ \mathtt{fst}\ \{\mathtt{v}_1, \mathtt{v}_2\} \mid \nu \mid t \to \mathtt{v}_1 \mid \nu \mid t \qquad \text{(E-PairSnd)}\ \ \mathtt{snd}\ \{\mathtt{v}_1, \mathtt{v}_2\} \mid \nu \mid t \to \mathtt{v}_2 \mid \nu \mid t$$

$$\text{(E-Pair1)}\quad \frac{\mathtt{e}_1 \mid \nu \mid t \to \mathtt{e}_1' \mid \nu' \mid t + 1}{\{\mathtt{e}_1, \mathtt{e}_2\} \mid \nu \mid t \to \{\mathtt{e}_1', \mathtt{e}_2\} \mid \nu' \mid t + 1} \qquad \text{(E-Pair2)}\quad \frac{\mathtt{e}_2 \mid \nu \mid t \to \mathtt{e}_2' \mid \nu' \mid t + 1}{\{\mathtt{v}_1, \mathtt{e}_2\} \mid \nu \mid t \to \{\mathtt{v}_1, \mathtt{e}_2'\} \mid \nu' \mid t + 1}$$

4.3 Additional Typing Rules for Image Manipulation

Operations that manipulate images may explicitly specify a range of valid input sizes (*i.e.*, size constraints) on their input to ensure correct processing. In the example below, the face count opcode requires that its input be in the size range of 320 to 1024 (using manageable image widths as a size rather than actual total numbers of pixels which fall 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. This alternate approach has a distinct advantage over using T-WEAKEN prior to T-OPAPPLY (as is done in the Conditional in Section 3.5.3); namely the size and cost bounds are more accurate if they are computed against the input's actual size bound instead of weakened size bounds. It should be obvious to the reader that the *latentcost* values for the functions in this section have also been contrived in a manner to ease the presentation and discussion.

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

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

A resampling operation is analogous to casting an image to be of a different size.[5] It has no explicit values for the size variables of its input. Its only constraint is that the input be of some positive size (greater than zero).

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

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

Image sensing hardware (*e.g.*, a web camera) has the capability to capture images in a range of possible resolutions. While the sized annotation makes sizing explicit in the type system, our programming language lacks explicit type annotation within its syntax. Thus, we formally define a new primitive to allocate a new image sensor (*ala* `image`) that accepts a resolution range explicitly, as an argument.

$$\text{(T-SIZEDIMAGE)}\quad \frac{\mathit{type}(\texttt{sizedimage}) = \tau \rightarrow \mathit{Sensor}\ \mathit{Img}^{\{r_1,r_2\}} \qquad \mathit{constr}(\texttt{sizedimage}) = \kappa_1}{\texttt{sizedimage} : \tau \rightarrow \mathit{Sensor}\ \mathit{Img}^{\{r_1,r_2\}}\ \$\ 0, \kappa_1}$$

$$\begin{aligned} \tau &= \mathit{Int}^{\{n_1,n_2\}} \times \mathit{Int}^{\{n_3,n_4\}} \\ \kappa_1 &= \{n_1 > 0, n_3 > 0, r_1 \geq \mathit{min}(n_1, n_3), r_2 \leq \mathit{max}(n_2, n_4)\} \\ \mathit{latentcost}(\texttt{sizedimage}, \tau) &= (\mathit{maxsize}(\tau)/8) \end{aligned}$$

[5] 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 Section 5.

4.4 Flowtypes

Finally we introduce a new function whose sole purpose is to inject run-time constraints (a Flowtype in our nomenclature). This function annotates that its argument has an explicit deadline within the type system. For example, the example with function `period()` from the Introduction would be implemented as syntactic sugar using `deadline()`.

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

$$\text{(E-DEADLINE)}\quad (\mathtt{deadline\ n\ e}_2 \mid \nu \mid t) \rightarrow (\mathtt{e}_2 \mid \nu \mid t)$$

4.5 In Practice

In this section we have expanded our core language as much as possible so as to reflect our real operating and tasking environment. We will now construct several examples on which we apply our type system.

Inferring Optimal Image Resolution

In practice, a user may not provide a specific resolution (size) bound for images that are used as part of a larger computation in which the image is not part of the desired output. 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. As image sensors are able to capture images in a range of possible resolutions, 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.

Example 1:

Face counting within an image from an image sensor with no explicit size bounds: `facect(get(image))`

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

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

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

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.

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). As a result we advocate the use of the maximum size, provided there are resources available to accommodate the additional processing overhead. This processing overhead is easily measured by the cost function (c_2, in the above).

Example 2:

Face counting within an image from an image sensor with no explicit size bounds *with an explicit deadline:* `deadline` 322 `facect` (`get`(`image`))

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

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

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

It is worth noting that, despite slight differences in the typing derivation, this example has a size bound result that is identical to an example with explicit (user-specified) size bounds that are the same as those imposed by the `facect` opcode (`deadline` 322 `facect`(`get`(`sizedimage`({320, 1024})))).

Example 3:

Face counting within an image from an image sensor with no explicit size bounds *as the response in a* ***trigger*** *expression:* `trigger` $\mathtt{e}_0$ `facect(get(image))`

We assume the presence of a Boolean typed expression $\mathtt{e}_0$ with cost c_0 and constraint set κ_0.

$$\frac{\mathtt{e}_0 : \mathit{Bool}\ \$\ c_0, \kappa_0 \qquad \mathcal{D}_1}{\mathtt{trigger}\ \mathtt{e}_0\ \ \mathtt{facect(get(image))} : \mathit{Int}^{\{n_1,n_2\}}\ \$\ c_3, \kappa_0 \cup \kappa_1 \cup \kappa_2}\text{(T-Trig)}$$

$$\begin{aligned} c_1 &= 1 + \mathit{latentcost}(\mathtt{get}, r_2) = 1 + (r_2/8) \\ c_2 &= 1 + c_1 + \mathit{latentcost}(\mathtt{facect}, r_2) = 1 + c_1 + r_2/2 \\ \kappa_1 &= \{r_1 \geq \mathtt{r}_{min}, r_2 \leq \mathtt{r}_{max}\} \\ \kappa_2 &= \{r_1 \geq 320, r_2 \leq 1024\} \\ c_3 &= (\mathcal{T} * c_0) + c_2 \end{aligned}$$

If expression $\mathtt{e}_0$ starts running at time t_{start}, transitions to `true` at time t_{true}, and $\mathit{delay}(\mathtt{e}_0)$ is the time to evaluate $\mathtt{e}_0$ then we can write: $\mathcal{T} = \lceil \frac{t_{true} - t_{start}}{\mathit{delay}(\mathtt{e}_0)} \rceil$. Recognizing that c_0 is a delay (bound) for $\mathtt{e}_0$, we can express this as $\mathcal{T} = \lceil \frac{t_{true} - c_2}{c_0} \rceil$. This example underscores the correctly bounding (or estimating) the "weight" of $\mathcal{T}$ is dependent on estimating the factors required to cause $\mathtt{e}_0$ to transition to true, which may include information about the sensing environment $\mathcal{E}$ and costs determined elsewhere in the derivation (*i.e.*, to supply t_{start}).

Example 4:

Face counting within a *resampled* image, originally captured from an image sensor with no explicit size bounds: `facect` (`resample` {4, `get(image)` })

$$\mathcal{D}_2 = \frac{4 : \mathit{Int}^{\{n,n\}}\ \$0, \{n = 4\} \qquad \mathcal{D}_0}{\{4, \mathtt{get(image)}\} : \mathit{Int}^{\{n,n\}} \times \mathit{Img}^{\{r_1,r_2\}}\ \$\ c_1, \kappa_1 \cup \{n = 4\}}\text{(T-PairCons)}$$

$$\mathcal{D}_3 = \frac{\mathit{type}(\mathtt{resample}) = \mathit{Int}^{\{n,n\}} \times \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Img}^{\{r_3,r_4\}} \qquad \mathit{constr}(\mathtt{resample}) = \kappa_2}{\mathtt{resample} : \mathit{Int}^{\{n,n\}} \times \mathit{Img}^{\{r_1,r_2\}} \rightarrow \mathit{Img}^{\{r_3,r_4\}}\ \$\ 0, \kappa_2}\text{(T-Resample)}$$

$$\mathcal{D}_4 = \frac{\mathcal{D}_2 \qquad \mathcal{D}_3}{\mathtt{resample}\ \{4,\ \mathtt{get(image)}\ \} : \mathit{Img}^{\{r_3,r_4\}}\ \$\ c_2, \kappa_1 \cup \kappa_2 \cup \{n = 4\}}\text{(T-OpApply)}$$

$$\frac{\mathit{type}(\mathtt{facect}) = \mathit{Img}^{\{r_3,r_4\}} \rightarrow \mathit{Int}^{\{n_3,n_4\}} \qquad \mathit{constr}(\mathtt{facect}) = \kappa_1}{\dfrac{\mathtt{facect} : \mathit{Img}^{\{r_3,r_4\}} \rightarrow \mathit{Int}^{\{n_3,n_4\}}\ \$\ 0, \kappa_3 \qquad \mathcal{D}_4}{\mathtt{facect}\ (\mathtt{resample}\ \{4,\ \mathtt{get(image)}\ \}) : \mathit{Int}^{\{n_3,n_4\}}\ \$\ c_3, \kappa_4}\text{(T-OpApply)}}\text{(T-FaceCt)}$$

$$
\begin{aligned}
c_1 &= 1 + \mathit{latentcost}(\texttt{get}, r_2) = 1 + (r_2/8) \\
c_2 &= \mathit{latentcost}(\texttt{resample}) + c_1 \\
c_3 &= \mathit{latentcost}(\texttt{facect}) + c_2 \\
\kappa_1 &= \{r_1 \geq \texttt{r}_{min}, r_2 \leq \texttt{r}_{max}\} \\
\kappa_2 &= \{n > 0, r_1 > 0, r_3 = r_1 * 4, r_4 = r_2 * 4\} \\
\kappa_3 &= \{r_3 \geq 320, r_4 \leq 1024\} \\
\kappa_4 &= \{n = 4\} \cup \kappa_1 \cup \kappa_2 \cup \kappa_3
\end{aligned}
$$

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.

Worst-Case Computational Cost and Expression 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 a scalar value 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 other Flowtypes) can not be met as specified. As presented, the `deadline` opcode relies on the unadjusted (discretized) computation bound.

In addition to static verification, the Service Dispatcher component of snBench often must 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.

4.6 Implementation Details

The type system described in this document has been implemented as part of our snBench project. The implementation has been done in Java, making use of the open-source JavaCC project [Sri]. In its current incarnation it is presented as part of the snBench development tool chain; the type checker is automatically invoked when compiling our high-level language (SNAFU) programming language to STEP. As the implementation is entirely modular and checks STEP code rather than SNAFU code, nothing stops us from using the type checker *apart from* the SNAFU compiler, however (1) SNAFU is eminently more readable than STEP and (2) the implementation includes several hooks to track line numbers within SNAFU code as an error tracking convenience to its users. Put differently, while the type system can exist without SNAFU, we have gone to some lengths to ensure SNAFU users can enjoy the benefit of this type checker as well.

Our implementation of the type checking engine includes a broader range of STEP constructs than the Core STEP that is presented here, however there are some limitations to this support; for constructs for which we lack the complete formalism use heuristics to bound cost and default back to basic, unsized type checking rather that annotated type checking for opcodes which lack size constraint annotation.

To solve the constraint sets that are built as a result of the sized type checking, we invoke the GNU Linear Programming Toolkit (GLPK) [And], which 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 external script, though nothing (other than time) precludes finer integration via the GLPK Java Interface [Bjo].

Figures 2 and 3 are screen shots taken from the the SNAFU development environment that feature results generated from the implementation of the type checker.

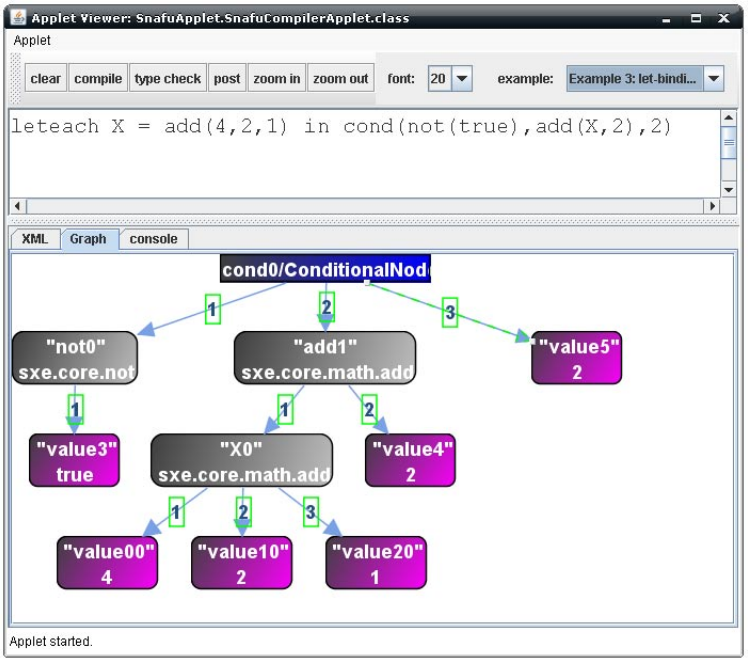

Figure 2: A screen shot from the SNAFU compiler showing the successful type checking of a STEP program. By default, the feedback is given graphically to the user.

5 Future Work

5.1 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 larger ordered set and defining the desired subtyping relation.

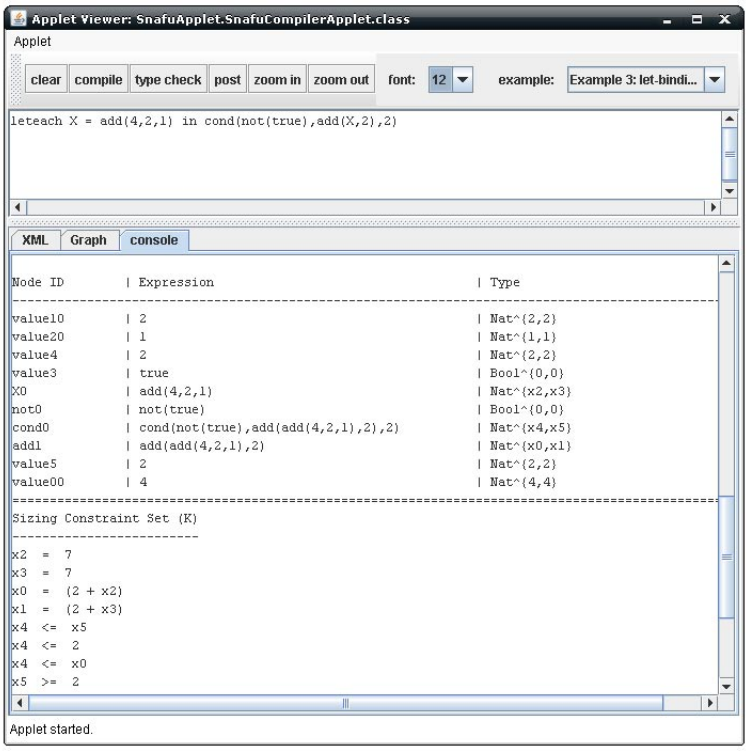

Figure 3: A screen shot from the SNAFU compiler showing size annotation constraints that result from type checking a STEP program.

One such example is the notion of image quality, which is different from data size. Data *size* is used 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 size (resolution) of the image changes no longer reflects the number of data points it contained originally (we are calling 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 an image's lower size bound.[6] This value could distinguish between a true high resolution image and data that has been up-cast or coerced to satisfy a function's 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.

[6]Recall the definition of T-Resample increases both the upper and the lower size bound.

5.2 Applications to Image Pyramids

Image Pyramids [Kro91],[AH91] 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 the appropriate resolution can be selected for manipulation depending on the needs of the manipulation function. The applications of this formalism to this 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.

5.3 Cost and Size Signatures 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).

5.4 Different Models of Cost

At present our type system provides a single model of computational cost, a worst-case upper bound. As there is the possibility for multiple size annotations (dimensions) of types, there is an opportunity to provide multiple cost metrics based on these other dimensions. Defining a so-called "minimum" computational cost (or optimal case) would be trivial, and other cost models including non-computation costs models (*e.g.*, defining some financial cost, or total memory utilization cost) would be possible, if not trivial. The approach that seems the most straight forward to enabling multiple cost models would be to provide functional costs, such that all costs computed on the right hand side of the $ would be user/programmer customizable and configurable as a function of the data available in the typing rules.

5.5 Expanding to "Complete" STEP

At present we support only a subset of the STEP programming language. The remaining STEP constructs (*e.g.*, streaming/persistent triggers and their associated read semantics) are difficult to represent as a result of their asynchronous execution and complex internal state. Despite these challenges, we have worked to establish a typing formalisms for these constructs however they are presently incomplete. Despite this, the present limited form is useful for non-persistent, "straight-shot" programs (which actually capture a significant amount of image processing tasks). For persistent/streaming trigger constructs we apply a simple heuristic: we solve the computable portion and multiply by estimates to derive the approximate costs for services that include non-Core portions of STEP. Regardless of this makeshift solution, it would be advantageous (and naturally appealing) to have a type system that includes the complete STEP language.

6 Conclusions

In this paper, we have presented our formal type system for multi-dimensional sized types. 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

[AH91] F. Ackermann and M. Hahn. Image pyramids for digital photogrammetry. *Digital Photogrammetric Systems*, pages 43–59, 1991.

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

[BBKO05] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Michael 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.

[Bjo] Bjoern Frank. GLPK 4.8 Java Interface. http://bjoern.dapnet.de/glpk/index.htm.

[Bos] Boston University, Department of Computer Science. Sensorium Research Homepage. http://www.cs.bu.edu/groups/sensorium/.

[HFH06] Kevin Hammond, Christian Ferdinand, and Reinhold Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Rev.*, 3(4):27–36, 2006.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[Kro91] 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.

[LH96] Hans-Wolfgang Loidl and Kevin Hammond. A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

[OBK06] Michael J. Ocean, Azer Bestavros, and Assaf J. Kfoury. SNBENCH: Programming and Virtualization Framework for Distributed Multitasking Sensor Networks. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 89–99, New York, NY, USA, 2006. ACM Press.

[RG94] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

[Sri] Sriram Sankar, Metameta. Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. https://javacc.dev.java.net/.