A Multivalued Language with a Dependent Type System

Neal Glew
Intel Labs
aglew@acm.org

Tim Sweeney
Epic Games
tim.sweeney@epicgames.com

Leaf Petersen
Intel Labs
leaf.petersen@intel.com

Abstract

Type systems are used to eliminate certain classes of errors at compile time. One of the goals of type system research is to allow more classes of errors (such as array subscript errors) to be eliminated. Dependent type systems have played a key role in this effort, and much research has been done on them. In this paper, we describe a new dependently-typed functional programming language based on two key ideas. First, it makes no distinction between expressions, types, kinds, and sorts—everything is a term. The same integer values are used to compute with and to index types, such as specifying the length of an array. Second, the term language has a multivalued semantics—a term can evaluate to zero, one, multiple, even an infinite number of values. Since types are characterised by their members, they are equivalent to terms whose possible values are the members of the type, and we exploit this to express type information in our language. In order to type check such terms, we give up on decidability. We consider this a good tradeoff to get an expressive language without the pain of some dependent type systems. This paper describes the core ideas of the language, gives an intuitive description of the semantics in terms of set-theory, explains how to implement the language by restricting what programs are considered valid, and sketches the core of the type system.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Dependent types; multivalued term language; Type:Type

1. Introduction

Strong static type systems eliminate errors. Compilers refuse to compile programs that fail the type discipline, guaranteeing that those programs that pass the discipline are free of a certain category of errors. For mainstream languages that category includes applying primitive operations to values of inappropriate type, calling something that is not a function, subscripting something that is not an array, and so on. However, it does not include errors such as out-of-bounds subscripting or creating red-black trees that do not satisfy the red-black invariant—these errors can only be checked dynamically; eliminating them statically is desirable.

DTP '13, September 24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2384-0/13/09…$15.00.
http://dx.doi.org/10.1145/2502409.2502412

Dependent type systems can be used to eliminate such errors. Traditional dependent type systems replace the usual pair and function types with dependent pair and function types. For example, instead of the usual pair type $\tau_1 * \tau_2$, a dependent type system might have a dependent pair type of the form $\Sigma x : \tau_1.\tau_2$ where τ_2 can refer to the first component of the pair using x, and similarly for dependent function types. In full generality, if an expression e has such a dependent pair type then its second projection, $e.2$, has type $\tau_2\{x \mapsto e.1\}$. Such generality raises two obvious questions: what does this mean if e can have effects, and how does recursion at the expression level interact with such a type construct? One way to answer the former question is to ban effects and have only a pure language. This approach works fine for a logic or a theorem prover, but a practical programming language must incorporate effects in some form. Another approach is to restrict dependency to values rather than general expressions, and we take an approach along these lines.

For the latter question, our approach is simply to give up on the decidability of the type system. Most past proposals for dependently typed languages have had decidability of the type system as a goal, and so were forced to take a different approach. DML [24] and later systems [23] achieve dependency by using singleton types and polymorphism. They add integers and operations on them to the type level, and an integer sort to the kind level to classify them. Then the same integer-kinded type variable can be used in two places to express dependency between the two things being typed, with universal and existential quantifiers to introduce the type variable. DML also uses constrained polymorphism to add constraints between type variables allowing more expressive dependency. The drawback to this approach is that integers are duplicated at the type level. Dependency through other kinds of values would also require those kinds to be duplicated at the type level. The programmer may also, depending upon how much inference can be done, need to explicitly write the introduction and elimination of polymorphism, which can become burdensome.

Coq [10], Epigram [1], and Agda [19] all have traditional dependent type systems, but are pure, total, and require the programmer to supply explicit proofs in some circumstances. These choices are quite appropriate for a logic based on the propositions-as-types principle and theorem provers for them, or for a programming language with statically checked termination. However, it can be difficult to convince the type checker that something is total, and the additional explicit annotations can be burdensome to the programmer. Idris [5] is similar to these systems but is not strictly total. It includes partiality, totality assertions, postulates, total checking, and IO and other effects via monads, which give the programmer more flexibility and results in a more practical programming language. It is also a strict language, as is ours.

Our long term goal is a practical programming language targetting programmers for whom the additional effort of writing extensive annotations and programming at the type level would be unacceptable. We do not claim to have achieved this goal yet, but describe the key ideas behind a new approach that anecdotal experience with prototypes suggests might have the right properties. In particular, we present a new dependent type system that represents a new point in the design space from previous work. Our design uses two key ideas, neither of which is new, but how we use them to make a dependent type system is, we believe, novel.

The first idea is that we have a single term language rather than separate languages for expressions, types, kinds, and sorts. Systems like the λ-cube [3] and λ-calculus with Type:Type [6] are based on such collapsed term languages and our system has some notable similarities (particularly with the latter). One important difference is that instead of one construct for functions, λ, and a separate construct for types, Π, our language uses the same construct for both functions and function types. This idea goes back at least to Nederpelt [18], but is not common in modern languages. Similarly we use the same construct for both pairs and pair types. The same integers are used at runtime and for singleton types, array lengths, etc. We can compute with types just as we compute with values. This single language gives our system conceptual simplicity, and we believe is easier for programmers to understand.

The second idea is that terms are multivalued—they can evaluate to more than one value, including none or even an infinite number. We take the idea from Ontic [15, 16], but similar ideas appear in other work. In Ontic, terms describe interesting sets of values, and many set-theoretic constructs are easily expressed. However, Ontic is not a type system and its goals are different. In other settings, multivaluedness is used for computation, making search and backtracking implicit and allowing the programmer to concentrate on expressing what to search for. In our language, runtime computation essentially happens only with single values and building more interesting sets is instead used to build the type system. To the best of our knowledge, this is a novel idea.

Of course, there is no free lunch and this generality comes at a cost, mainly that our system is not decidable. It is possible (actually quite easy) to write programs that will cause static checking to diverge. Other mainstream type systems (e.g. Hindley-Milner), have non-polynomial time complexities, and checking pathological programs in those systems is for all practical purposes indistinguishable from divergence. The key question for our purposes, is whether the programs programmers wish to write fall into the decidable fragment of the type system, and how easy it is for them to fall into the undecidable fragment by accident. Based on our initial experiences with prototype systems, we believe our language does well on these criteria. Other systems have made this choice before us—Cayenne [2] is the first dependent type system we know of with undecidable type checking.

We begin the paper by describing how some examples are expressed in our system to motivate what follows. Then we informally illustrate a core term language, $\lambda_\aleph$, still using examples. These examples show what we mean by a multivalued language, how we use multivaluedness to represent types, and how we ultimately build up a dependent type system. We formalise $\lambda_\aleph$ by giving a denotational semantics that we call the set-theoretic semantics. Unfortunately, the formalisation has an unresolved technical problem, so we show what we have so far. Also, as explained below, this semantics is not implementable for all terms. Instead of using it directly, we define an implementation called the runtime for a subset of the full term language, formalised as a small-step operational semantics. A static process, called the verifier, checks that a program is in the language subset the runtime is intended to implement. Our language is strongly typed, and the verifier is also responsible for type checking—ensuring the absence of runtime type errors. We informally describe how the verifier works.

Rather than reduce our core language to a minimum and precisely describe it, we have chosen to use a more comprehensive core language and spend much of the paper explaining the key underlying ideas and intuitions, leaving less space to precisely describe our system. We intend this paper to serve as an overview of a new idea, rather than a detailed formalisation. Thus we only sketch many parts of the set-theoretic semantics, runtime, and verifier, and leave a complete formalisation to future papers.

1.1 Other Related Work

There is abundant related work on dependent type systems, of which the following discussion only touches on a small selection. Refinement type systems (e.g. Dminor [4]) start with a conventional type system and add refinements, subseting types in a set-theoretic sort of way; our system similarly builds types up in a set-theoretic sort of way, but has a single level rather than two levels. HMC [11] is based on refinement types and infers refinement predicates by observing that these are relations on the variable of the refined type and the current in-scope variables; one can view our multivalued terms that represent types as similarly being relations on the value of type being defined and in-scope variables, and thus based on their observation, but their system and ours are built differently. YTT and Ynot [17] combine dependent typing, logic, and effects in the same language by separating programs into pure and computational parts; like DML they have duplication of constructs and rules across levels. Trellys [13] also combines dependent typing with general recursion, restricting to values like us; they have multiple levels and are not yet working on a practical source language. Sage [14] also targets a practical language with more error checking, employing a single term language, refinement types, and hybrid checking—using dynamic checks when static checks fail—their system is more conventional than ours, and we seek fully static checks. F* [21] targets securely verifying program properties in a practical langauge; it has value-based dependent types and integrates well with the .NET framework. Aura [12] is a dependently-typed language targeted at security authorisation and has a single term language. Our use of set-theoretic constructs to build types and some of the way that our type checking works resembles the set-theoretic approaches to subtyping [7, 8], but the two are significantly different. A very recent work [22] attempts to make Haskell into a more dependently-typed language, notably they collapse the kind and term levels.

2. Dependent-Typing Examples

An important motivating example of dependent typing, and one in which we are particularly interested, is array-bounds checking. We build up to an extended example by first considering simply a function that takes a length and an in-bounds index. In our language that could be written as:[1]

$$f(n : \mathsf{Nat}, i : \mathsf{Nat} < n) = \cdots$$

This example defines f to be a function with two parameters, n and i. The syntax $: \mathsf{Nat}$ is a term, and terms specify sets of values, in this case, the natural numbers.[2] Writing $n : \mathsf{Nat}$ says that n may be bound to any value in the set specified by $: \mathsf{Nat}$, that is, that n may be any natural number. This means that f may only be called with an actual first argument that is a natural number. Similarly the term $: \mathsf{Nat} < n$ specifies a set of values. In this case it refers to n and it specifies the set of natural numbers that are less than n. Thus f may only be called with an actual second argument that is a natural number and that is less than the first argument. If n is the length

[1] This section will use a hypothetical surface syntax that is based on actual prototype languages we have built. The following sections will describe a core language that is substantially desugared.

[2] The next section will explain in detail why the terms used in the examples specify the sets we claim.

of some array then i is an in-bounds index for that array. The $\cdots$ above is also some term, and if f is a function called at runtime[3] then it should be a term that specifies exactly one value, namely, the value the function should compute at runtime when called.

Now consider changing this example to take an array and subscript it. That could expressed in our language as:

$$g(a : [\,]\mathsf{Int}, i : \mathsf{Nat} < \mathsf{len}(a)) = a(i)$$

Here the term $: [\,]\mathsf{Int}$ specifies the set of arrays of integers of any length. Now i ranges over natural numbers that are less than the length of a, namely, in-bounds indices for a. Array subscript and function application use the same syntax in our language and are statically verified, in this case, to be free at runtime of the out-of-bounds error.

Next consider elementwise addition of two arrays of integers. That can be expressed as:

$$\begin{aligned}&add(a_1 : [\,]\mathsf{Int}, a_2 : [\mathsf{len}(a_1)]\mathsf{Int}) =\\ &\quad \mathsf{arr}[\mathsf{len}(a_1)]i \mapsto (a_1(i) + a_2(i))\end{aligned}$$

Here the term $: [\mathsf{len}(a_1)]\mathsf{Int}$ specifies the set of arrays of integers of length $\mathsf{len}(a_1)$ and $\mathsf{arr}[\mathsf{len}(a_1)]i \mapsto (a_1(i) + a_2(i))$ computes an array of length $\mathsf{len}(a_1)$ initialising element i to $a_1(i) + a_2(i)$. In this example, both subscripts $a_1(i)$ and $a_2(i)$ are verified as in bounds, add can only be called with arrays of the same length, and the type checker knows the result is an array of that length.

We can also use dependency in data structures to capture some invariants. Consider for example a compressed sparse row representation of square matrices. Such a representation consists of an array of the non-zero entries of the matrix, an array giving the columns of these entries, and an array giving for each row and a sentinel the starting position in the other arrays. We could express that as:

$$\begin{aligned}&\mathsf{record}\{n : \mathsf{Nat}, nnz : \mathsf{Nat}, nzs : [nnz]\mathsf{Int},\\ &\quad c : [nnz]\mathsf{type}\{:\mathsf{Nat} < n\},\\ &\quad rs : [n+1]\mathsf{type}\{:\mathsf{Nat} \le nnz\}\}\end{aligned}$$

Here the field n is the size of the matrix, nnz is the number of non-zero entries, nzs those entries, c the columns for the entries, and rs the row starting positions; $: [nnz]\mathsf{type}\{:\mathsf{Nat} < n\}$ should be read as all arrays of length nnz of natural numbers less than n. The example precisely specifies the lengths of the arrays nzs, c, and rs, and that c and rs have entries in very specific ranges.

Finally consider multiplying a matrix m from the above type with a dense vector v of type $[m.n]\mathsf{Int}$, where $m.n$ denotes the selection of field n from the matrix m. Entry i of that product is the dot product of row i of m with v, row i starts at $rs(i)$ in nzs and c and ends before $rs(i+1)$; nzs gives the value of the entry and c the column, i.e. which entry of v to multiply with. Thus we can compute the product as:

$$\begin{aligned}&\mathsf{arr}[m.n]i \mapsto\\ &\quad \mathsf{sum}\ j = m.rs(i)\ \mathsf{to}\ m.rs(i+1) - 1\ \mathsf{of}\\ &\qquad m.nzs(j) * v(m.c(j))\end{aligned}$$

All the subscripting in this example statically verifies as in bounds, and the type checker knows the result has type $[m.n]\mathsf{Int}$.

Hopefully these examples give some intuition as to how our language expresses dependent types. Whereas in conventionally typed languages the programmer specifies the values a variable may have by writing a type, and types specify sets of values, in our language the programmer writes a term, and terms specify sets of values. Programmers also specify computations by writing terms, and the term language is as expressive as expression languages in conventional functional languages. These brief examples and explanations probably raise more questions than they answer about our language, so in the next section we build up a core term language using a series of examples to explain how the various term constructs work, what sets of values they specify, and how they can be used to encode constructs from conventional type systems.

3. Multivalued Language $\lambda_\aleph$ by Example

The driving intuition behind $\lambda_\aleph$ is that every term can be thought of as describing or specifying a set of values. In this section we introduce the language by example using an informal $values[\![t]\!]$ function which gives the set of values specified by a term t.

Basics The most basic terms are primitive integer terms, such as the term 3. As a term, 3 specifies a set containing only a single value—specifically the integer 3. Hence we say that $values[\![3]\!] = \{3\}$. As usual, there are builtin operations on the integers—hence $3 + 4$ describes only the integer 7, that is, $values[\![3 + 4]\!] = \{7\}$. More generally, most terms with multiple subterms have a cross product semantics, taking any combination of the values of the subterms. Hence, for a term $t_1 + t_2$ we have $values[\![t_1 + t_2]\!] = \{x + y \mid x \in values[\![t_1]\!] \wedge y \in values[\![t_2]\!]\}$.

Join A more interesting term is $3 \mid 4$, which specifies either 3 or 4, that is, $values[\![3 \mid 4]\!] = \{3, 4\}$. In general join gives us union of values, so $values[\![t_1 \mid t_2]\!] = values[\![t_1]\!] \cup values[\![t_2]\!]$. Using the join construct combined with integer addition, we can build up larger sets; for example, $values[\![(3 \mid 4) + (5 \mid 7)]\!] = \{8, 9, 10, 11\}$.

Unify We express intersection of sets of values using a construct called unify, written $t_1 == t_2$. For example, $values[\![(3 \mid 4) == (4 \mid 5)]\!] = \{4\}$. In general $values[\![t_1 == t_2]\!] = values[\![t_1]\!] \cap values[\![t_2]\!]$.

Variables and lets A key design decision in the language is that while the term language is multivalued, variables are always single valued. A term t specifies a set of values $values[\![t]\!]$—however, binding t to a variable x does *not* bind x to the set $values[\![t]\!]$, but rather binds x to each of the elements of $values[\![t]\!]$ in turn. For terms which specify only a single element, this behaves much like variable binding. For example, the meaning of $\mathsf{let}(x = 4)(x+3)$ is $\{7\}$, since the values of 4 include only 4, and so the values of $x+3$ include only the values which arise from replacing x with 4. In general, the meaning of the term $\mathsf{let}(x = t_1)t_2$, can be thought of as the union of the results of binding x to each value of $values[\![t_1]\!]$ in turn, taking the meaning of t_2 under that binding. So for example, the meaning of the term $\mathsf{let}(x = 3 \mid 4)(x + x)$ includes the values 6 and 8 but not 7, since the meaning of $x+x$ is taken with respect to a single choice of binding for x taken from the set $\{3, 4\}$ rather than allowing the choice to be made anew at each use of the variable.

No values An important point is that in addition to specifying multiple values, a term can also specify no values. We give primitive syntax for such a term since it is particularly useful. The term falses has no values in its meaning, that is, $values[\![\mathsf{falses}]\!] = \{\}$. Note this is unrelated to divergence and type errors—falses is neither divergent nor erroneous.

Conditions Instead of representing boolean truth values directly, we instead use multivaluedness, taking the idea of inhabited types representing true propositions literally—a term with at least one value is called inhabited[4] and represents logical true, and a term with no values, like falses, is called uninhabited and represents logical false. The usual logical operations can then be encoded using existing constructs. Join gives us disjunction—$t_1 \mid t_2$ is inhabited iff either t_1 or t_2 is inhabited; and pairs (see below) give us

[3] Later we will see that functions are also used to express function types, and that such functions would not be called at runtime.

[4] Technically, a term with at least one outcome is inhabited, see below for a discussion of outcomes.

conjuction—(t_1, t_2) is inhabited iff both t_1 and t_2 are inhabited because of the cross product semantics.

Conditionals Given this definition of conditions, the primitive conditional construct in $\lambda_\aleph$ simply tests for inhabitance. For example, in $\mathsf{if}\ (x = 10)\ 1\ \mathsf{else}\ 2$ the condition 10 is inhabited and so the meaning of the conditional is the meaning of the true branch, yielding 1; whereas in $\mathsf{if}\ (x = \mathsf{falses})\ 1\ \mathsf{else}\ 2$ the condition falses is uninhabited and so the meaning of the conditional is the meaning of the false branch, yielding 2. The conditional construct also binds a variable to each possible value of the condition when it is inhabited for use in the true branch, collecting up the results using union as with the let construct.

Comparisons Given our definition of conditionals in terms of inhabitance, the unify construct already gives us a generic notion of equality. For example, $3{=}{=}3$ is inhabited (with 3), whereas $3{=}{=}4$ is uninhabited. Other comparisons for integers must be defined primitively. For example, we define $t_1 < t_2$ as specifying all of the integers in t_1 which are less than at least one of the integers in t_2. Of course, if no such integer exists, then the comparison specifies an empty set. So the term $3 < 4$ specifies the set $\{3\}$, whereas $4 < 3$ is uninhabited, and $values[\![(3 \mid 5) < 4]\!] = \{3\}$ while $values[\![(3 \mid 5) < (4 \mid 6)]\!] = \{3, 5\}$.

Infinite values As well as terms which specify finitely many values, there are also terms whose meaning is an infinite number of values. For example, the term ints specifies the set of all integers, $values[\![\mathsf{ints}]\!] = \mathbb{I}$. Terms such as this one can be combined with comparisons to yield terms which specify useful subsets of the integers. The term $\mathsf{ints} \geq 0$ has the meaning $\mathbb{N}$, and hence specifies the natural numbers. Further, if we define $\mathsf{nats} = (\mathsf{ints} \geq 0)$, then we can construct a set $values[\![\mathsf{nats} < 10]\!] = \{0, \ldots, 9\}$ that gives a term describing the in-bounds indices of an array of length 10. Another important infinite valued term is the anys term which specifies the set of all values.

Arrays There are terms for fixed-length tuples which produce record values mapping indices to values. Thus the term $(2, 3)$ specifies the set containing the value $\langle 0 \mapsto 2, 1 \mapsto 3\rangle$. Fixed-length tuples also have cross-product semantics, so $values[\![(3 \mid 4, 5 \mid 6)]\!] = \{\langle 0 \mapsto 3, 1 \mapsto 5\rangle, \langle 0 \mapsto 3, 1 \mapsto 6\rangle, \langle 0 \mapsto 4, 1 \mapsto 5\rangle, \langle 0 \mapsto 4, 1 \mapsto 6\rangle\}$. Projection from tuples is denoted syntactically via application to indices. So the term $(3, 4)(0)$ specifies $\{3\}$ and $(3, 4)(1)$ specifies $\{4\}$. Application of tuples to indices are always statically checked, and hence out-of-bound indexing such as $(3, 4)(2)$ is a type error.

There are also terms that specify arrays of dynamic length. The term $\mathsf{arr}[10]i \mapsto i + 11$ specifies the singleton set containing the length 10 array $\langle 0 \mapsto 11, \ldots, 9 \mapsto 20\rangle$. Note there is no difference between a fixed-length tuple and an array. Both are instances of a more general notion of tables. Also note that in general the length is a term, and the type checker verifies that it is a natural number.

Of course, with dynamically-sized arrays, it may not always be practical to statically check that indices are always within the bounds of the table. The statically-checked application form used above, $t_1(t_2)$, gives a type error if t_2 is not in the domain of t_1, and hence we refer to it as *error application*. In contrast, a second application form $t_1[t_2]$ has the property that it is uninhabited (but not an error) if t_2 is not in the domain of t_1. We refer to this as *failing application*. Intuitively, error application corresponds to an unchecked subscript, in which the burden of proving that the index is in the domain falls to the static checker, whereas failing application corresponds to a checked subscript, in which a dynamic check of some sort must be done at runtime.

Tables Arrays and tuples are special cases of a more general construct called tables, which are finite mappings from values to values. Tuples and arrays are special cases of tables which map the indices $\{0, \ldots, n - 1\}$ to values where n is the number of elements in the tuple or array. More general tables are possible and useful in many programming scenarios, but we do not discuss these further except as necessary to discuss the encoding of discriminated unions.

Discriminated unions To encode discriminated unions in $\lambda_\aleph$, we use a form of tables with a single domain value. For example, to encode $\mathsf{in_L}(3)$ in $\lambda_\aleph$, we use the table term $(0 \mapsto 3)$, which specifies the table $\langle 0 \mapsto 3\rangle$. Similarly, to encode $\mathsf{in_R}(7)$ we use the table term $(1 \mapsto 7)$, which specifies the table $\langle 1 \mapsto 7\rangle$. The encoding of the case construct relies on the failing-application form discussed above. Using failing application to project out of the table, we can see that the meaning of $(0 \mapsto 3)[0]$ is 3 whereas $(0 \mapsto 3)[1]$ is uninhabited. Similarly, $(1 \mapsto 7)[0]$ is uninhabited and the meaning of $(1 \mapsto 7)[1]$ is exactly 7. Since conditionals branch on inhabitance, we can thus encode $\mathsf{case}(t)\ \mathsf{in_L}(x_1).t_1 \mid \mathsf{in_R}(x_2).t_2$ as $\mathsf{let}(x = t)\mathsf{if}\ (x_1 = x[0])\ t_1\ \mathsf{else}\ \mathsf{let}(x_2 = x(1))t_2$ for fresh x.

Dependent Tuples Our terms for fixed-length tuples allow for expressing dependency. In general they take the form $(x_1 = t_1, \ldots, x_n = t_n)$ where term t_j can refer to previous entries using the variables $x_1, \ldots, x_{j-1}$. So $(x = 3, y = x + 1)$ specifies the single table $\langle 0 \mapsto 3, 1 \mapsto 4\rangle$, and $(x = 3 \mid 4, y = x + 1)$ specifies the two tables $\langle 0 \mapsto 3, 1 \mapsto 4\rangle$ and $\langle 0 \mapsto 4, 1 \mapsto 5\rangle$.

Functions As with other terms, function terms specify sets of function values—however, there are some surprising subtleties to the interpretation of function terms. The intuitive idea is that function values serve to map values to values, not values to sets of values. For functions whose bodies are single-valued, this distinction is largely unimportant. For example, the term $\mathsf{fn}^\circ(x = 3)x + 1$ specifies the function value whose domain is $\{3\}$ and that maps 3 to 4. Similarly, $\mathsf{fn}^\circ(x = \mathsf{ints})x + 1$ specifies the function value whose domain is the integers and that maps any given integer to that integer plus one. For each of these examples, there is at least informally exactly one function value that each function term evaluates too, but function terms can be multivalued too. For example, $\mathsf{fn}^\circ(x = \mathsf{ints})\mathsf{ints}$ specifies the set containing every function value whose domain is the integers and that maps integers to integers (and there are many such functions). The key point is that this last term does not specify a function which "returns" the set of all integers, but rather specifies a set containing all of the functions with the given domain and range. This distinction is especially noticeable with terms such as $\mathsf{fn}^\circ(x = \mathsf{ints})\mathsf{falses}$, which rather than specifying a function which returns no result, is instead interpreted as being uninhabited, that is, $values[\![\mathsf{fn}^\circ(x = \mathsf{ints})\mathsf{falses}]\!] = \{\}$.

A second key property of functions in $\lambda_\aleph$ is that a function value has an explicit domain corresponding to the set of values to which the function may be applied, and for each value in the domain, application of the function produces some outcome: either a value, divergence, or a type error. For values outside of the domain of a function value, the result of applying the function to such a value depends on the kind of application used. As with tables, the error application form commits a type error if the argument is outside of the domain of the function; whereas the failing application form is uninhabited if the argument is outside of the domain.

The term language has several kinds of function terms which differ in the precise set of function values that they specify. The examples above use the term form that we refer to as *invariant functions*. In general, an invariant function term has the form $\mathsf{fn}^\circ(x = t_1)t_2$ and specifies all of the the functions whose domain is exactly $values[\![t_1]\!]$ and that maps any value in this set to a value from the meaning of t_2 with x bound to that domain value. We refer to this class of functions as invariant functions since t_1 specifies exactly the domain of the function values that the term specifies.

In contrast, the terms for *contravariant functions* specify only a lower bound on the domain. These function terms have the form $\mathsf{fn}^-(x = t_1)t_2$ and specify all of the functions whose domains are a superset of $values[\![t_1]\!]$ and that map values in $values[\![t_1]\!]$ to a value in the meaning of t_2 with x bound to that domain value (without specifying anything about how the function value behaves on arguments outside of $values[\![t_1]\!]$). Contravariant function terms are generally used for normal programming in $\lambda_\aleph$ as they obey the usual contravariant argument subtyping rule. Invariant functions are used for expressing types.

Type representations Intuitively, types are often thought of as sets of values and terms in our multivalued language essentially specify sets of values. So at an informal level, one can imagine using $\lambda_\aleph$ terms as representations of types. For example, the empty type which classifies no values is represented by the term falses; and the type of all values is represented by the term anys. Singleton types such as $\mathsf{S}(3)$ are represented simply by literal terms, in this case 3. The base integer type can be represented by ints, and ranges of integers can be represented using comparisons—for example, nats, $\mathsf{nats} < 10$, and $\mathsf{ints} \geq 5 \leq 20$. Similarly, if t_1 represents τ_1 and t_2 represents τ_2 then (t_1, t_2) represents the pair type $\tau_1 * \tau_2$ and $(0 \mapsto t_1) \mid (1 \mapsto t_2)$ represents the sum type $\tau_1 + \tau_2$. If t represents the type τ and n is a natural number then $\mathsf{arr}[n]i \mapsto t$ represents the type of arrays of length n whose elements have type τ (where i is fresh); similarly, $\mathsf{arr}[\mathsf{nats}]i \mapsto t$ represents arrays of type τ (where i is fresh). If t_1 represents τ_1 and t_2 represents τ_2 and x does not occur in t_2 then $\mathsf{fn}^-(x = t_1)t_2$ represents the non-dependent function type $\tau_1 \to \tau_2$.

Thus, all first-order types that make sense for our language can be represented as terms in the language. More interesting types can be represented as well. For example, $(x_1 = \mathsf{ints}, x_2 = \mathsf{ints} < x_1)$ represents the type of all pairs of integers where the second is less than the first. Similarly, $\mathsf{fn}^-(x = \mathsf{ints})\mathsf{ints} < x$ represents the type of functions taking integers to integers less than the argument. Quite a number of interesting dependent product and function types are expressible in our term language.

Types The representations of most interesting types make use of multivalued terms. In order to support computation over types it is necessary to be able to encode type representations as single values. For example, while the term $\mathsf{fn}^-(x = \mathsf{ints})\mathsf{ints}$ represents the set of all integer to integer functions, introducing a name for ints via naive variable binding does not do the right thing. The term $\mathsf{let}(y = \mathsf{ints})\mathsf{fn}^-(x = y)y$ no longer represents the type of all integer to integer functions, but rather the set of all functions whose domain is a single integer i and which map i to i.

The solution to this problem is to "wrap" types up as a single value in the term language employing a simple trick similar to previous work using retractions to define types [6]. In the case of $\lambda_\aleph$, a type is an identity function whose domain is the values of the type. For example, $\mathsf{fn}^\circ(x = \mathsf{ints})x$ is a term that specifies exactly one value, the function whose domain is the integers and that maps each integer to itself. The syntactic sugar $\mathsf{type}\{t\}$ is shorthand for $\mathsf{fn}^\circ(x = t)x$ where x is fresh. Thus $\mathsf{type}\{\mathsf{falses}\}$ is a single value for the empty type, $\mathsf{type}\{3\}$ is a single value for the singleton 3 type, $\mathsf{type}\{\mathsf{nats}\}$ is a single value for the type of natural numbers, and so on. The Int of the previous section is actually definable in our language as $\mathsf{type}\{\mathsf{ints}\}$.

To make types useful, there are two further term forms. The term types specifies the set of all types,[5] that is, all of the function values that are identity functions. Syntactic sugar Type stands for $\mathsf{type}\{\mathsf{types}\}$, namely the type of types. Finally, the term $:t$ "unwraps" the encoding by extracting out the domains of all of the identity functions specified by t; if t includes something that is not an identity function then $:t$ gives a type error. We call this operation "from", and it allows us to convert a type back into a multivalued term from the members of the type.

[5] If we have a third kind of function then we can define types, but to keep things simple, we will leave types as a primitive term.

Using these constructs, we can now bind type representations to names with the correct semantics by wrapping and unwrapping types. For example, the term $\mathsf{let}(y = \mathsf{type}\{\mathsf{ints}\})\mathsf{fn}^-(x = :y):y$ specifies the set of all integer to integer functions, as desired.

Type computation Since types can be encoded as values in the language, they can be computed on in a completely general fashion. As a simple example, the function

$$\mathsf{fn}^-(x = (\mathsf{types}, \mathsf{types}))\mathsf{type}\{(:x(0), :x(1))\}$$

takes any two types and returns the type of pairs whose components have those two types. Specifically, x will be bound to a pair of types, $x(0)$ is the first of these, $:x(0)$ any element of it, $(:x(0), :x(1))$ is a pair of an element from the first type and an element of the second type, and wrapping that with type turns it into the type of such pairs. In a similar fashion, it is possible to write any type constructor from typical $F_{\underline{\omega}}$ like languages (that is, F_ω without polymorphism—see Section 7 for a discussion of polymorphism). The previous section used the prefix operator $[\,]$, which is definable as $\mathsf{fn}^-(t = \mathsf{types})\mathsf{type}\{\mathsf{arr}[\mathsf{nats}]i \mapsto :t\}$—namely, it takes a type and returns the type of all arrays whose elements are in that type.

Programs Finally, our intention is to use multivaluedness only to specify type information, and a program in our core language is a term that specifies a singleton set, namely the single value the programmer wants computed. Similarly, the functions that are called at runtime must have bodies that specify a single value. Terms that specify sets of zero or one value arise in the condition subterms of conditionals, but terms that specify sets of other cardinalities are used only to describe type information and are never run.

4. Set-Theoretic Semantics

The examples of the previous section are intended to show the potential of $\lambda_\aleph$—combining in a useful way traditional typing with a set-theoretic intuition and potentially enabling practical dependent typing. In this section, we start formalising $\lambda_\aleph$. We do not have enough space to fully describe this formalisation and it is in any case not yet completely figured out, so we will show only some of the details of what we have so far. This section defines the set-theoretic semantics, a kind of denotational semantics, which makes clear what a term means. This semantics is not directly implementable, so in the next section we explain how to select a sublanguage that is implementable and how to realise that implementation.

As with any denotational semantics, we need to define semantic domains that serve as the meaning of terms and define a meaning function that maps a term to its meaning. The main semantic domains for $\lambda_\aleph$ are values and outcomes, defined below. The main meaning functions are *outcomes* and *isin*. The first maps a term to a set of outcomes. The second tests whether a given value is one of the values specified by a given term. It cannot just be defined in terms of *outcomes* because, as explained more below, it must compute the test and might diverge while doing so. The definitions of these functions requires some machinery to present clearly. Rather than develop all that machinery and show the full two definitions, instead we will present a restriction of *outcomes* to values—the function *values* that we used informally in the previous section. This function is much easier to describe, and additionally the intuition of $\lambda_\aleph$ is actually much more apparent in its definition since *outcomes* and *isin* are somewhat obscured by details of evaluation sequencing. The derivation of this function from *outcomes* is made

$$
\begin{array}{lcl}
\mathit{values}\llbracket x\rrbracket_\rho & = & \{\rho(x) \mid x \in \mathit{dom}(\rho)\} \\
\mathit{values}\llbracket \mathsf{falses}\rrbracket_\rho & = & \{\} \\
\mathit{values}\llbracket \mathsf{anys}\rrbracket_\rho & = & \{v \mid v \in V\} \\
\mathit{values}\llbracket i\rrbracket_\rho & = & \{i\} \\
\mathit{values}\llbracket \mathsf{ints}\rrbracket_\rho & = & \{i \mid i \in \mathbb{I}\} \\
\mathit{values}\llbracket \mathit{uop}\ t\rrbracket_\rho & = & \{\mathit{uop}\ i \mid i \in \mathit{values}\llbracket t\rrbracket_\rho\} \\
\mathit{values}\llbracket t_1\ \mathit{bop}\ t_2\rrbracket_\rho & = & \{i_1\ \mathit{bop}\ i_2 \mid i_1 \in \mathit{values}\llbracket t_1\rrbracket_\rho \wedge i_2 \in \mathit{values}\llbracket t_2\rrbracket_\rho\} \\
\mathit{values}\llbracket t_1\ \mathit{cop}\ t_2\rrbracket_\rho & = & \{i_1 \mid i_1 \in \mathit{values}\llbracket t_1\rrbracket_\rho \wedge i_2 \in \mathit{values}\llbracket t_2\rrbracket_\rho \wedge i_1\ \mathit{cop}\ i_2\} \\
\mathit{values}\llbracket (x_1 = t_1, \ldots, x_n = t_n)\rrbracket_\rho & = & \{\langle 0 \mapsto v_1, \ldots, n-1 \mapsto v_n\rangle \mid \forall 1 \le j \le n : v_j \in \mathit{values}\llbracket t_j\rrbracket_{\rho\{x_1 \mapsto v_1, \ldots, x_{j-1} \mapsto v_{j-1}\}}\} \\
\mathit{values}\llbracket (i \mapsto t)\rrbracket_\rho & = & \{\langle i \mapsto v\rangle \mid v \in \mathit{values}\llbracket t\rrbracket_\rho\} \\
\mathit{values}\llbracket \mathsf{arr}[t_1]x \mapsto t_2\rrbracket_\rho & = & \{\langle 0 \mapsto v_1, \ldots, n-1 \mapsto v_n\rangle \mid \\
 & & \quad n \in \mathit{values}\llbracket t_1\rrbracket_\rho \wedge \forall 1 \le j \le n : v_j \in \mathit{values}\llbracket t_2\rrbracket_{\rho\{x \mapsto j-1\}}\} \\
\mathit{values}\llbracket \mathsf{tabs}\rrbracket_\rho & = & \{\mathit{st} \mid \mathit{st} \in \mathit{Tb}\} \\
\mathit{values}\llbracket \mathsf{fn}^\circ(x = t_1)t_2\rrbracket_\rho & = & \{\mathit{sf} \mid \forall v \in V : \mathit{sf}.1(v) = \mathit{isin}(t_1, \rho, v) \wedge \\
 & & \qquad (\mathit{isin}(t_1, \rho, v) = \mathsf{id} \implies \exists o \in O : \mathit{sf}.2(v) = o \wedge o \in \mathit{outcomes}\llbracket t_2\rrbracket_{\rho\{x \mapsto v\}})\} \\
\mathit{values}\llbracket \mathsf{fn}^-(x = t_1)t_2\rrbracket_\rho & = & \{\mathit{sf} \mid \forall v \in V : \mathit{isin}(t_1, \rho, v) = \mathsf{id} \implies \\
 & & \qquad \mathit{sf}.1(v) = \mathsf{id} \wedge \exists o \in O : \mathit{sf}.2(v) = o \wedge o \in \mathit{outcomes}\llbracket t_2\rrbracket_{\rho\{x \mapsto v\}}\} \\
\mathit{values}\llbracket \mathsf{funs}\rrbracket_\rho & = & \{\mathit{sf} \mid \mathit{sf} \in F\} \\
\mathit{values}\llbracket \mathsf{types}\rrbracket_\rho & = & \{\mathit{ty} \mid \mathit{ty} \in \mathit{Ty}\} \\
\mathit{values}\llbracket \mathsf{len}(t)\rrbracket_\rho & = & \{n \mid \mathit{sa} \in \mathit{values}\llbracket t\rrbracket_\rho \wedge \mathit{sa} \in A \wedge \mathit{dom}(\mathit{sa}) = \{0, \ldots, n-1\}\} \\
\mathit{values}\llbracket t_1(t_2)\rrbracket_\rho & = & \quad \{v \mid \mathit{st} \in \mathit{values}\llbracket t_1\rrbracket_\rho \wedge \mathit{st} \in \mathit{Tb} \wedge v_2 \in \mathit{values}\llbracket t_2\rrbracket_\rho \wedge v_2 \in \mathit{dom}(\mathit{st}) \wedge v = \mathit{st}(v_2)\} \\
 & & \cup\{v \mid \mathit{sf} \in \mathit{values}\llbracket t_1\rrbracket_\rho \wedge \mathit{sf} \in F \wedge v_2 \in \mathit{values}\llbracket t_2\rrbracket_\rho \wedge \mathit{sf}.1(v_2) = \mathsf{id} \wedge v = \mathit{sf}.2(v_2)\} \\
\mathit{values}\llbracket {:}t\rrbracket_\rho & = & \{v \mid \mathit{ty} \in \mathit{values}\llbracket t\rrbracket_\rho \wedge \mathit{ty} \in \mathit{Ty} \wedge \mathit{ty}.1(v) = \mathsf{id}\} \\
\mathit{values}\llbracket t_1 \mid t_2\rrbracket_\rho & = & \mathit{values}\llbracket t_1\rrbracket_\rho \cup \mathit{values}\llbracket t_2\rrbracket_\rho \\
\mathit{values}\llbracket t_1 == t_2\rrbracket_\rho & = & \mathit{values}\llbracket t_1\rrbracket_\rho \cap \mathit{values}\llbracket t_2\rrbracket_\rho \\
\mathit{values}\llbracket \mathsf{let}(x = t_1)t_2\rrbracket_\rho & = & \cup_{v \in \mathit{values}\llbracket t_1\rrbracket_\rho} \mathit{values}\llbracket t_2\rrbracket_{\rho\{x \mapsto v\}} \\
\mathit{values}\llbracket \mathsf{if}\ (x = t_1)\ t_2\ \mathsf{else}\ t_3\rrbracket_\rho & = & \begin{cases} \mathit{values}\llbracket \mathsf{let}(x = t_1)t_2\rrbracket_\rho & \mathit{outcomes}\llbracket t_1\rrbracket_\rho \ne \{\} \\ \mathit{values}\llbracket t_3\rrbracket_\rho & \mathit{outcomes}\llbracket t_1\rrbracket_\rho = \{\} \end{cases}
\end{array}
$$

Figure 1. The Value Part of the Set-Theoretic Semantics

precise in the second subsection below, but Figure 1 can be viewed informally as providing its definition. Rather than give an explicit grammar for $\lambda_\aleph$, we leave the grammar implicit in Figure 1.

4.1 Semantic Domains

In most languages, an expression evaluates to a value, evaluates to a type error, or diverges—we calls these possibilities *outcomes*. In $\lambda_\aleph$, the meaning of a term is a set of outcomes. In $\lambda_\aleph$, a *value* is either an integer, a table, or a function. Note that tables and functions are disjoint. A *table* has a domain that is, for this paper, a finite subset of the integers and has, for each domain integer, a corresponding range value. A *function* has a domain that is a subset of the values and for each domain value has a corresponding range outcome. Thus, our semantic domains are something like:

$$
\begin{array}{lcl}
V & = & \mathbb{I} + (\mathbb{I} \overset{\text{fin}}{\to} V) + (V \rightharpoonup O) \\
O & = & V + \{\perp, \mathsf{err}\}
\end{array}
$$

In standard set theory, these equations have no solution as $V \rightharpoonup O$ has greater cardinality than V. The usual approach is to seek CPOs and restrict to continuous functions; domain theory then has standard techniques for solving equations like those above. However, for a long time we still could not find a satisfactory overall semantics. Recently, we realized a key insight—namely, that testing if a value is in a function's domain should be computable, that is, such tests rather than evaluating just to in or out, might instead diverge or commit a type error. Thus we should model our semantic functions as pairs of a domain-test function and a range funtion. The domain-test function takes a value and returns a member of $\{\perp, \mathsf{err}, \mathsf{nid}, \mathsf{id}\}$, where $\perp$ means the test diverges, err means the test commits an error, nid means the value is not in the domain, and id means the value is in the domain. The range function takes a value and returns an outcome, that in the case the value is in the domain is the result of applying the function. These modifications lead to these equations:

$$
\begin{array}{lcl}
V & = & \mathbb{I} + \mathit{Tb} + F \\
\mathit{Tb} & = & \mathbb{I} \overset{\text{fin}}{\to} V \\
F & = & [V \to \mathit{DT}] \times [V \to O] \\
\mathit{DT} & = & \{\perp, \mathsf{err}, \mathsf{nid}, \mathsf{id}\} \\
O & = & V + \{\perp, \mathsf{err}\}
\end{array}
$$

We solve these equations using slightly modified techniques from domain theory—we wish to have cyclic structures and so we model tables using regular infinite trees rather than finite trees as is standard. Unfortunately, these modifications are not enough. Some syntactic function terms that we expect to be inhabited in fact have no semantic functions in their meaning because the semantic functions we expect to be there are in fact not continuous. Resolving this technical issue remains to be solved.

For the rest of this section we ignore this issue and proceed as if the above were the right semantic domains, since we feel that the meaning functions serve to illuminate the intended semantics of the language even if the domains are not yet quite right. Let A be the subset of Tb that are arrays, that is, have domains of the form $\{0, \ldots, n-1\}$. Let Ty be the subset of F that are types, that is, whose range function is $\lambda v.v$.

Terms have free variables and so their meaning is relative to an environment, which for $\lambda_\aleph$, are finite maps from variables to values reflecting both which variables are in scope and the single-valued nature of variables. Metavariable ρ ranges over environments of the set-theoretic semantics.

4.2 Value Semantics

As mentioned previously, the main definitions of meaning are the *outcomes* and *isin* functions. If t is a term and ρ is an environment then $outcomes[\![t]\!]_\rho$ is a subset of O, namely t's outcomes. If t is a term, ρ is an environment, and v is a value then $isin(t, \rho, v)$ is a member of DT. Having defined these functions, we define $values[\![t]\!]_\rho = outcomes[\![t]\!]_\rho \cap V$, namely the restriction of t's meaning to just the values. Figure 1 can then be shown to be a theorem (as opposed to an informal definition).

Figure 1 uses a number of metavariables: x ranges over variables, t over terms, i over integers, n over natural numbers, uop over unary operations on the integers, bop over binary operations on the integers, cop over comparison operators (of the integers), v over V, o over O, st over Tb, sa over A, sf over F, and ty over Ty.

Both error and failing application have the same value semantics, so we show only error application, however, their meanings differ in error behaviour. Also note that conditionals test absence of outcomes not absence of values—if the condition diverges then so does the conditional.

5. Runtime

The set-theoretic semantics presented in the previous section is very intuitive, but unfortunately, it is not realisable on a conventional computer. For starters, the semantics of $t_1 == t_2$ implies that the implementation can decide the equality of functions, something that is undecidable. It is also unclear how to implement multivaluedness in general. One could use backtracking in some cases, or generate lists of result values, but all of these options are more expensive than a conventional single-valued language. We intend multivaluedness to express type information, but not actual computation, and would rather not pay an implementation cost to support multivaluedness. Additionally, it is not clear how multivaluedness should interact with computational effects like mutation and IO. Thus the implementation of the language does not fully follow the set-theoretic semantics but instead implements a subset of the full term language. We call the actual implementation of the language, the *runtime*, and in this section we formalise the runtime. One should think of this formalisation as an operational semantics for the language.

The overall strategy is to exploit a static analysis called the verifier to reject programs that cannot be effectively realised, allowing the runtime to only address the set of programs satisfying the invariants enforced by verification. As with a standard type checker, the verifier is responsible for detecting and rejecting type errors in programs. However, the verifier must also detect and reject programs which cannot be realised by the runtime. Most importantly, these include programs that potentially compare function values, or require runtime computation of terms with more than one value.

Of course, not all uses of multivaluedness should be rejected. There are several key uses of multivalued terms that must be supported, and these uses drive much of the design of the verifier and runtime. In the first place, terms which do not need to be run (such as those serving in the role of types) can be multivalued. Second, our conditionals, comparisons, and encoding of sum types are fundamentally based on the distinction between an inhabited and uninhabited term. Thus the runtime must support the special case of terms that have zero or one values. This is easily implementable in a natural way using exceptions. A term with zero values is said to fail, and causes a failure exception to be raised. Conditionals in turn catch this exception and execute the false branch of the conditional.

Finally, some uses of terms of more than one value can be given a reasonable runtime interpretation. Consider for example the term $t_1 == t_2$. If we were to fully restrict this to single valued terms, we might compute its value by computing single values for each of t_1 and t_2 and then comparing these two values for equality. However, we can achieve greater expressiveness by only requiring that one of the two terms be single valued in many cases. For example, having computed a single value v for t_1, it is often feasible to test whether v is a member of the (potentially) multiple values specified by t_2, using the structure of t_2 to guide the membership test. For example, if t_2 is the term ints we can just check that v is an integer. For the term $t_{21} \mid t_{22}$ we can check if v is a value of t_{21} and if not check if it is a value of t_{22}. For the term (t_{21}, t_{22}) we can first check that the value is a pair, and if so check that the respective components of the pair are possible values of t_{21} and t_{22} respectively. There are limits to this strategy of course. For more general terms, we must fall back to requiring a single value for t_2. For example, to check if v is a value of $\mathsf{let}(x = t_{21})t_{22}$ we need to first compute a single value for t_{21}, bind that to x, and then check if v is a member of the (again possibly multivalued) term t_{22}. Finally, for terms such as $t_{21}(t_{22})$, we really have no choice but to compute a single value for this term and do an equality test.

This special treatment of unification motivates a key point in the design of the runtime. In particular, the runtime has two modes called generate and test. Each subterm of the program is statically classified as a generate term, a test term, or an unevaluated term. Generate terms must be zero or one valued, and the verifier will reject the program if they are not. Test terms can have more than one value, but the verifier must still check that no function comparison is done. Unevaluated terms can be arbitrary.

In the rest of this section, we formalise the runtime as an abstract machine with a small-step reduction relation. Since this is mostly straightforward and there are many details, we only show a few reduction rules to illustrate how they operate. A complete and formal treatment of the runtime can be found in an accompanying technical report [9].

5.1 Abstract Machine

The abstract machine of the runtime consists of three parts: a term to be evaluated, an environment that maps the term's free variables to *head labels*, and a head heap that maps head labels to *heads*. Heads are the value form for terms and have three forms: integers, tables, and functions. Sub-components of heads are indirected via head labels which index into the head heap. The machine operates over a slightly extended set of terms, which include head labels, a frame form, and a test mode form which is explained further below. The syntax of the abstract machine is as follows:

$$\begin{array}{lcl}
t & ::= & \cdots \mid \ell \mid \sigma(t) \mid \ell \in t_1 \rhd t_2 \ \mathsf{else}\ t_3 \\
\sigma & ::= & x_1 = \ell_1, \ldots, x_n = \ell_n \\
dk & ::= & - \mid \circ \\
h & ::= & i \mid \langle i_1 \mapsto \ell_1, \ldots, i_n \mapsto \ell_n \rangle \mid (\sigma, \mathsf{fn}^{dk}(x = t_1)t_2) \\
HH & ::= & \ell_1 = h_1, \ldots, \ell_n = h_n \\
M & ::= & (HH; \sigma; t)
\end{array}$$

5.2 Machine Reduction Rules

The reduction rules for the abstract machine have the forms $M_1 \mapsto M_2$, which means that M_1 steps to M_2 in one step; $M \mapsto \mathsf{F}$, which means that M fails in the next step; and $M \mapsto \mathsf{E}$, which means that M commits a type error in the next step. As there are many rules for the language we treat here, and most of these are standard, we present, in Figure 2, only a few of the novel rules to illustrate the novel aspects such as failure, failure application, and test mode. All the rules can be found in the technical report.

Generate Mode A few points are worth noting about generate mode. The term falses fails immediately. The terms anys, ints, tabs, funs, types, $:t$, and $t_1 | t_2$ might have more than one value, and so reduce to a type error in generate mode. Failure application has two interesting cases. If the LHS is a table and the RHS is not in the domain of the table then it fails. If the RHS is an invariant function, then the runtime must first check if the RHS is in the domain of that function. It does this using a test-mode term, explained shortly.

$$\frac{}{(HH;\sigma;\mathsf{falses}) \mapsto \mathsf{F}} \qquad \frac{}{(HH;\sigma;\mathsf{anys}) \mapsto \mathsf{E}}$$

$$\frac{HH(\ell_1) = i_1 \quad HH(\ell_2) = i_2 \quad i_1\ cop\ i_2}{(HH;\sigma;\ell_1\ cop\ \ell_2) \mapsto (HH;\sigma;\ell_1)} \qquad \frac{HH(\ell_1) = i_1 \quad HH(\ell_2) = i_2 \quad \neg(i_1\ cop\ i_2)}{(HH;\sigma;\ell_1\ cop\ \ell_2) \mapsto \mathsf{F}}$$

$$\frac{HH(\ell) = i \quad i \geq 0 \quad y \notin fv(t)}{(HH;\sigma;\mathsf{arr}[\ell]x \mapsto t) \mapsto (HH;\sigma;(y = \mathsf{let}(x = 0)t, \ldots, y = \mathsf{let}(x = i-1)t))}$$

$$\frac{HH(\ell) = \langle 0 \mapsto \ell_1, \ldots, n-1 \mapsto \ell_n \rangle}{(HH;\sigma;\mathsf{len}(\ell)) \mapsto (HH;\sigma;n)} \qquad \frac{HH(\ell_1) = \langle i_1 \mapsto \ell'_1, \ldots, i_n \mapsto \ell'_n \rangle \quad HH(\ell_2) = h \quad h \notin \{i_1, \ldots, i_n\}}{(HH;\sigma;\ell_1[\ell_2]) \mapsto \mathsf{F}}$$

$$\frac{HH(\ell_1) = (\sigma', \mathsf{fn}^{\circ}(x = t_1)t_2)}{(HH;\sigma;\ell_1[\ell_2]) \mapsto (HH;\sigma;\sigma'(\ell_2 \in t_1 \rhd \mathsf{let}(x = \ell_2)t_2\ \mathsf{else}\ \mathsf{falses}))}$$

$$\frac{}{(HH;\sigma;{:}t) \mapsto \mathsf{E}} \qquad \frac{}{(HH;\sigma;t_1 \mid t_2) \mapsto \mathsf{E}} \qquad \frac{}{(HH;\sigma;\ell == t) \mapsto (HH;\sigma;\ell \in t \rhd \ell\ \mathsf{else}\ \mathsf{falses})}$$

$$\frac{}{(HH;\sigma;\mathsf{if}\ (x = \ell)\ t\ \mathsf{else}\ t') \mapsto (HH;\sigma;(\sigma, x = \ell)(t))} \qquad \frac{(HH;\sigma;t_1) \mapsto \mathsf{F}}{(HH;\sigma;\mathsf{if}\ (x = t_1)\ t_2\ \mathsf{else}\ t_3) \mapsto (HH;\sigma;t_3)}$$

$$\frac{}{(HH;\sigma;\ell \in \mathsf{falses} \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;t_2)} \qquad \frac{}{(HH;\sigma;\ell \in \mathsf{anys} \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;t_1)}$$

$$\frac{HH(\ell) = i}{(HH;\sigma;\ell \in \mathsf{ints} \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;t_1)} \qquad \frac{HH(\ell) = h \quad h \text{ not an integer}}{(HH;\sigma;\ell \in \mathsf{ints} \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;t_2)}$$

$$\frac{HH(\ell) = \langle i \mapsto \ell' \rangle}{(HH;\sigma;\ell \in (i \mapsto t) \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;\ell' \in t \rhd t_1\ \mathsf{else}\ t_2)}$$

$$\frac{HH(\ell') = (\sigma', \mathsf{fn}^{\circ}(x = t_3)t_4)}{(HH;\sigma;\ell \in {:}\ell' \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;\ell \in \sigma'(t_3) \rhd t_1\ \mathsf{else}\ t_2)}$$

$$\frac{}{(HH;\sigma;\ell \in t_3 \mid t_4 \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;\ell \in t_3 \rhd t_1\ \mathsf{else}\ \ell \in t_4 \rhd t_1\ \mathsf{else}\ t_2)}$$

$$\frac{}{(HH;\sigma;\ell \in t_3 == t_4 \rhd t_1\ \mathsf{else}\ t_2) \mapsto (HH;\sigma;\ell \in t_3 \rhd \ell \in t_4 \rhd t_1\ \mathsf{else}\ t_2\ \mathsf{else}\ t_2)}$$

Figure 2. Selected Reduction Rules

Finally, unifying a head label with another term reduces to a test-mode term to check that the LHS value is in the RHS term and if so to return the LHS value, and otherwise to fail.

Test Mode Test mode is formalised as the test term $\ell \in t_1 \rhd t_2$ else t_3. This term is generated by the runtime to check whether the value given by ℓ is one of the values the term t_1 specifies. In the case that it is, it should reduce to t_2, otherwise it should reduce to t_3.

We show a few cases to illustrate how this works. Testing against falses always fails whereas against anys always succeeds. Testing against ints just tests whether the value is an integer or not. Testing against $(i \mapsto t)$ first checks that the value has the head form $\langle i \mapsto \ell' \rangle$ and then tests value ℓ' against t. Testing one value against a from of a second value that is an invariant identity function, that is, a type, tests the first value against the domain term of the type. Testing against a unification tests against the LHS term and if that succeeds tests against the RHS term otherwise fails. Similarly, testing against a join tests against the LHS term and if that fails tests against the RHS term otherwise succeeds.

Program Evaluation A program evaluates according to these rules, which show the initial machine and that programs cannot have zero values.

$$\frac{(\epsilon;\epsilon;t) \mapsto^* (HH;\epsilon;\ell) \quad HH(\ell) = i}{t \Downarrow_{\mathsf{R}} i} \qquad \frac{(\epsilon;\epsilon;t) \mapsto \cdots}{t \Uparrow_{\mathsf{R}}}$$

If neither rule applies then $t \Downarrow_{\mathsf{R}} \mathsf{E}$.

6. Verifier

$\lambda_{\aleph}$ is a statically typed language and a static checking process checks for and rejects programs that might commit type errors at runtime. Since the runtime only correctly implements a subset of the term language, this static checking process must also check for and reject programs not in the subset addressed by the the runtime—that is, the subset where the runtime and set-theoretic semantics agree. This static checking process is called the *verifier*. The verifier checks for and rejects type errors including traditional ones (applying integers, adding functions) and out-of-bounds subscripting errors. The verifier also checks for and rejects programs which compare functions, or in which terms of more than one value occur while in generate mode.

We start by informally describing how the examples in Section 2 are checked for type errors. Then we describe the structure of the verifier, giving examples to motivate each piece, describe a number of the details, and, in particular, describe set containment, a key component of the verifier.

6.1 Examples

To a first approximation the verifier makes one pass over the program computing along the way a representation of the set of values of each subterm. It uses these sets to check for errors that might arise at each subterm. Consider again the first example from Section 2:

$$f(n : \mathsf{Nat}, i : \mathsf{Nat} < n) = \cdots$$

In this example, the first $: \mathsf{Nat}$ will verify as free of errors and result in the set of values:

$$\{\ell_1 = \mathsf{ints}, \ell_2 = \ell_1 \geq 0\}.\ell_2$$

The meaning of this set is as follows: if ℓ_1 is bound to an integer, and ℓ_2 is bound to the same integer, and if that integer is greater than or equal to zero, then the integer bound to ℓ_2 is in the set; no other values are in the set. Thus the set describes exactly the natural numbers. This set will then be put into the context used to type check subsequent code with n bound to ℓ_2. The second $: \mathsf{Nat}$ similarly verifies with set:

$$\{\ell_3 = \mathsf{ints}, \ell_4 = \ell_3 \geq 0\}.\ell_4$$

The term n verifies with set:

$$\{\}.\ell_2$$

One should read this set as the singleton set of the value bound to ℓ_2 (by the context). Next, to check that $<$ does not commit an error, the verifier must verify that the two subterms are integers. It forms the set $\{\ell_5 = \mathsf{ints}\}.\ell_5$ describing all integers and asks if the sets of the two subterms are contained in this set. Since these sets describe integers, the verifier constructs formulas about integers and asks a theorem prover if those formulas are true. In this case it produces these formulas respectively (simplified):

$$\begin{array}{l} \forall i_3, i_4 \in \mathbb{I} : i_4 = i_3 \wedge i_4 \geq 0 \implies \exists i_5 \in \mathbb{I} : i_5 = i_4 \\ \forall i_1, i_2 \in \mathbb{I} : i_2 = i_1 \wedge i_2 \geq 0 \implies \exists i_5 \in \mathbb{I} : i_5 = i_2 \end{array}$$

Clearly both these formulas are true, so verification succeeds and produces the set (for the $<$ subterm):

$$\{\ell_3 = \mathsf{ints}, \ell_4 = \ell_3 \geq 0, \ell_6 = \ell_4 < \ell_2\}.\ell_6$$

This set describes the natural numbers less than ℓ_2. To type check the body of f, this set is also placed into the context and i bound to ℓ_6. Checking the body of the function might (for example) incur the requirement that i needs to be in bounds for an array of length n. In this case, the verifier will essentially have to check the following clearly true formula:

$$\begin{array}{l} \forall i_2, i_4, i_6 \in \mathbb{I} : \\ \quad i_2 \geq 0 \wedge i_4 \geq 0 \wedge i_6 = i_4 \wedge i_6 < i_2 \implies 0 \leq i_6 \wedge i_6 < i_2 \end{array}$$

Now consider the second example from Section 2:

$$g(a : [\,]\mathsf{Int}, i : \mathsf{Nat} < \mathsf{len}(a)) = a(i)$$

The term $: [\,]\mathsf{Int}$ verifies with set:

$$\{\ell_7 = \mathsf{ints}, \ell_8 = \ell_7 \geq 0, \ell_9 = [\ell_8]\ell_{10}.\{\ell_{11} = \mathsf{ints}\}.\ell_{11}\}.\ell_9$$

One should read this set as: if ℓ_8 is bound to a natural number, ℓ_9 is bound to an array of that length, and any element of the array is in the set $\{\ell_{11} = \mathsf{ints}\}.\ell_{11}$ when ℓ_{10} is bound to its index then that array is in the set. That is, it describes the set of arrays of integers of any length. Verifying the term $\mathsf{len}(a)$ requires checking that a is an array. The verifier forms the set

$$\{\ell_{12} = \mathsf{ints}, \ell_{13} = \ell_{12} \geq 0, \ell_{14} = [\ell_{13}]\ell_{15}.\{\ell_{16} = \mathsf{anys}\}.\ell_{16}\}.\ell_{14}$$

and asks if the previous set is contained in it which it is. The result of $\mathsf{len}(a)$ is described by the set:

$$\{\ell_{17} = \mathsf{len}(\ell_9)\}.\ell_{17}$$

Next, $: \mathsf{Nat} < \mathsf{len}(a)$ verifies with set:

$$\{\ell_{18} = \mathsf{ints}, \ell_{19} = \ell_{18} \geq 0, \ell_{17} = \mathsf{len}(\ell_9), \ell_{20} = \ell_{19} < \ell_{17}\}.\ell_{20}$$

To verify g's body, the verifier first verifies the subterms to get the sets $\{\}.\ell_9$ and $\{\}.\ell_{20}$. Next it must look at the possible values of ℓ_9 and confirm that each can be applied (are tables or functions) and that i is in their domain. In this case, ℓ_9 is an array, which is applicable. To do the domain test, the verifier forms a set describing the valid indices of ℓ_9:

$$\{\ell_{21} = \mathsf{ints}, \ell_{22} = \ell_{21} \geq 0, \ell_{23} = \ell_{22} < \ell_8\}.\ell_{23}$$

and asks if $\{\}.\ell_{20}$ is contained in this set, leading to the formula:

$$\begin{array}{l} \forall i_8, i_{17}, i_{18}, i_{20} \in \mathbb{I} : \\ \quad i_8 \geq 0 \wedge i_{17} = i_8 \wedge i_{18} \geq 0 \wedge i_{20} = i_{18} \wedge i_{20} < i_{17} \implies \\ \quad\quad \exists i_{23} : i_{23} \geq 0 \wedge i_{23} < i_8 \wedge i_{23} = i_{20} \end{array}$$

This formula is true.

6.2 Verifier Structure

Our verifier is similar to a conventional type checker but has a few twists. Such type checkers are often expressed as a set of syntax directed algorithmic rules, with one rule for each construct of the expression language. These rules take a typing context—a mapping of the variables in scope to their types—and output the type of the expression. In our language there is not a syntactic notion of type; instead we describe the set of values that each variable might take and that each term specifies. The language for describing these sets is quite rich, but not as expressive as the term language itself. We use a level of indirection in describing sets of values, much like the runtime does in describing single values. So a context might be $\ell_1 = \mathsf{ints}; x = \ell_1; \mathsf{T}$, which says that x is in scope and bound to some integer. Contexts also capture dependencies between variables, so $\ell_1 = \mathsf{ints}, \ell_2 = \ell_1 + 1; x = \ell_1, y = \ell_2; \mathsf{T}$ says that x is in scope and is some integer and y is in scope and is whatever x is plus one. Contexts also capture conditions that must hold, both through the value descriptions and through a side condition. For example, $\ell_1 = \mathsf{ints}, \ell_2 = \mathsf{ints}, \ell_3 = \ell_2 < \ell_1; x = \ell_1, y = \ell_3; \mathsf{T}$ says that x and y are integers and that y is less than x; and $\ell_1 = \mathsf{ints}, \ell_2 = 0; x = \ell_1; \ell_1 > \ell_2$ says that x is an integer and is greater than zero. The first form arises naturally from conditions in ifs and is used to type check true branches, whereas the second form is used to capture the negation of a condition while type checking false branches.

Similarly, the outputs of our rules describe sets of values in a similar way. For example, the term 5 would type check and output the set $\{\ell_1 = 5\}.\ell$. These output sets can depend upon the input contexts, for example, in the context $\ell_1 = \mathsf{ints}; x = \ell_1; \mathsf{T}$ the term $x + 1$ would type check and produce the output set $\{\ell_2 = 1, \ell_3 = \ell_1 + \ell_2\}.\ell_3$.

The rules in conventional type checkers follow a common pattern, for example, consider the rule for application for a non-dependent type system:

$$\frac{\Gamma \vdash f : \tau_1 \quad \vdash \tau_1 \leq \sigma_1 \to \sigma_2 \quad \Gamma \vdash a : \tau_2 \quad \vdash \tau_2 \leq \sigma_1}{\Gamma \vdash f(a) : \sigma_2}$$

Here both subexpressions are type checked. One expression is required to be a subtype[6] of a particular form of type (a function type), subtyping relations are required to hold between specific parts of the types (argument type a subtype of parameter type), and various pieces of types are combined (here just the function result type) to make the output type. Our rules follow a similar pattern except that the role of types is played by descriptions of sets of values; hence subtyping for us is subsets of values, a judgement we call *set containment*, which is the heart of the system and where most of the interesting properties get checked. The analogy to types does not fully hold: the analogous concept to requiring a type to have a particular structure via subtyping works out somewhat differently in our system, but we lack space to describe this in detail.

[6] Many systems use equality rather than subtyping, but we can think of equality as just a limited form of subtyping.

Of course, our system is dependent, and a typical application rule in a dependent type system might look more like this:

$$\frac{\Gamma \vdash f : \tau_1 \quad \vdash \tau_1 \leq \Pi x : \sigma_1.\sigma_2 \quad \Gamma \vdash a : \tau_2 \quad \vdash \tau_2 \leq \sigma_1}{\Gamma \vdash f(a) : \sigma_2\{x \mapsto a\}}$$

The function result type σ_2 depends upon the actual argument through variable x, so the output type is σ_2 with x replaced by the actual argument a. In our system, function types are actually functions themselves. To perform the substitution then, we essentially just apply the function and compute the result. This recomputation of the values of the body is very similar to type checking, but without any error checking, and the result is essentially analogous to running a piece of the program.

Thus, we can think of the verifier as having two modes, verification and running. Both modes do a certain amount of processing to determine the values the term specifies (and some other stuff like effects). Verification mode, in addition, does extra processing to check for errors. Every subterm of the program is verified exactly once, and might be run zero or more times.

Unfortunately, a problem arises with this strategy. A prime example is recursion. To check factorial (in surface syntax):

$$\texttt{fact}(n : \mathsf{Int}) = \mathsf{if}(n \leq 1)1 \;\mathsf{else}\; n * \texttt{fact}(n-1)$$

The verifier will check the body with `fact` bound to this function and n bound to an abstract integer. This checking will eventually get to $\texttt{fact}(n-1)$, which will run the body of this function with n bound to an abstract natural number, which will eventually run the body with n bound to an abstract positive number, and so on. Thus the verifier gets into an infinite loop.

To address this issue, we provide the programmer with a construct to cut the verifier off, called *stage*. In this example, the programmer would write in the surface syntax:

$$\texttt{fact}(n : \mathsf{Int}) : \mathsf{Int} = \mathsf{if}(n \leq 1)1 \;\mathsf{else}\; n * \texttt{fact}(n-1)$$

which desugars into (simplifying a bit):

$$\begin{array}{l}\texttt{fact} = \\ \mathsf{fn}^-(n = \mathsf{ints})\mathsf{S}(\mathsf{ints}, \mathsf{if}\;(\_ = n \leq 1)\;1\;\mathsf{else}\;n * \texttt{fact}(n-1))\end{array}$$

Note that what looks like a return type annotation (:Int) turns into the stage construct $\mathsf{S}(\mathsf{ints}, \ldots)$; type annotations on variable declarations act similarly. The verifier proceeds as before, but when it runs the body for the recursive call, it runs the left subterm of the stage, ints, and uses that as the result of the stage. This captures the fact that the result of the recursive call could produce any arbitrary integer. In this way the verifier is prevented from looping by cutting off the recursion with a sound approximation of the actual result of running the function. Of course, for this approximation to in fact be sound, the verifier must do some checking when it verifies a stage cosntruct.

More concretely, stage works as follows. The runtime treats $\mathsf{S}(t_1, t_2)$ as if it were simply t_2. The verifier on the other hand treats the stage as if it were simply t_1. This is correct so long as the values t_2 specifies are a subset of the values the verifier thinks t_1 specifies. Thus, when running the stage, the verifier just runs t_1 and uses that as the result. When verifying the stage, it verifies both t_1 and t_2, checks that t_2's set is contained in t_1's set, and uses t_1's set as the result. This treatment is similar to the conventional treatment of type annotations as in the rule:

$$\frac{\Gamma \vdash e : \tau' \quad \vdash \tau' \leq \tau}{\Gamma \vdash (e : \tau) : \tau}$$

Now, in making value set descriptions, the verifier cannot always be completely accurate, but must in some instances approximate the possible values. In most contexts this is fine so long as the approximation is an upper bound—that is the verifier computes a set that is a superset of the actual values that might be computed by the runtime. In some contexts however, an upper bound is not sufficient for correctness. For example, in an application we need to check that the actual argument is in the domain of the function being called. If we have an upper bound on the actual argument and a lower bound on the domain term of the function then it is sound to check set containment between these two sets, but if we only have an upper bound on the domain then the check is unsound. Rather than computing both a lower bound and an upper bound, the verifier instead computes a flag that says whether the set is exactly the values the term specifies or just an upper bound on that set. In certain places (e.g domain terms) the verifier requires the set computed to be exact, rejecting the program otherwise. This requirement limits the expressivity of domain terms.

As mentioned earlier, in addition to doing type checking the verifier must also reject programs the runtime does not correctly implement. Specifically, it must reject multivalued terms in generate positions and any function comparisons. To implement the former, the verification algorithm also takes a flag (called a runtime context), indicating whether the term is generated, tested, or not evaluated. Terms like ints, tabs, $:t$, and $t_1 \mid t_2$ are rejected if in the generate runtime context. For the latter the rules for unify and failing application must check, except in the unevaluated runtime context, for the possibility of function comparison and reject such subterms. This checking can be done using fairly crude syntactic checks.

To make computation of conditions as tight as possible, we need to know if the condition is inhabited or not. To do this we compute a *decidability* as well as a representation of its values. We say a term is *true* if it always is inhabited. We say a term is *false* if it always is uninhabited. We say a term is *decidable* if it might or might not be inhabited or we do not know which is the case. In running a conditional, if the condition subterm is true then we only run the true branch, it if is false we only run the false branch, and if it is decidable we run both branches and combine the results with join.

To motivate how we deal with integer information and conditions, let us make our factorial example just a little more interesting by restricting it to operate only on natural numbers:

$$\texttt{fact}(n : \mathsf{Nat}) : \mathsf{Nat} = \mathsf{if}(n \leq 1)1 \;\mathsf{else}\; n * \texttt{fact}(n-1)$$

The body of the function is checked in a context like:

$$\ell_1 = \mathsf{ints}, \ell_2 = 0, \ell_3 = \ell_1 \geq \ell_2; n = \ell_3; \mathsf{T}$$

The condition will verify with set:

$$\{\ell_7 = 1, \ell_8 = \ell_3 \leq \ell_7\}.\ell_8$$

When this set is uninhabited we know that $\ell_3 > 1$ must hold. So the false branch is checked in the context:

$$\ell_1 = \mathsf{ints}, \ell_2 = 0, \ell_3 = \ell_1 \geq \ell_2, \ell_9 = 1; n = \ell_3; \ell_3 > \ell_9$$

In typing the recursive application, the domain term will generate a set like:

$$\{\ell_{10} = \mathsf{ints}, \ell_{11} = 0, \ell_{12} = \ell_{10} \geq \ell_{11}\}.\ell_{12}$$

The argument will verify with set:

$$\{\ell_{13} = 1, \ell_{14} = \ell_3 - \ell_{13}\}.\ell_{14}$$

The verifier will then check that this set is contained in the previous set in the context above. That check results in checking the formula:

$$\begin{array}{l}\forall i_3, i_{14} \in \mathbb{I} : \\ i_3 \geq 0 \wedge i_3 > 1 \wedge i_{14} = i_3 - 1 \implies \\ \exists i_{12} \in \mathbb{I} : i_{12} \geq 0 \wedge i_{12} = i_{14}\end{array}$$

6.3 Set Containment

The set containment algorithm takes a context and two sets s_1 and s_2 and decides if the values described by s_1 are a subset of those described by s_2 in that context. This section informally describes a declarative version of set containment, consisting of three phases: splitting joins, structural matching, and integer formulas.

The first phase deals with joins in the context and s_1. If we want to know if a term like $3 \mid 4$ is contained in a term like nats, set containment will just generate a formula such as:

$$\forall i_1 \in \mathbb{I} : i_1 = 3 \vee i_1 = 4 \implies \exists i_2 \in \mathbb{I} : i_2 \geq 0 \wedge i_2 = i_1$$

which is true. Set containment only uses formulas for integers though, so joins of non-integers require a different treatment. So to check if $\mathsf{tabs} \mid \mathsf{ints}$ is contained in anys, set containment needs, in general, to consider each possibility in turn. The first phase, join splitting, selects a sequence of joins in the context or s_1 and considers as two separate subproblems, the join being the left possibility and the join being the right possibility. Being a declarative formulation, so long as some selection of a sequence of joins leads to success, set containment holds; an actual algorithm needs to find a way to select the right joins. Note that splitting joins can lead to an exponential number of subproblems to consider if joins are used indiscriminately or the actual algorithm is not carefully structured.

The second phase is to match parts of s_2 with s_1 and the context by structure. Consider if $\{\ell_1 = 3, \ell_2 = 4, \ell_3 = (\ell_1, \ell_2)\}.\ell_3$ is contained in $\{\ell_4 = \mathsf{ints}, \ell_5 = \mathsf{ints}, \ell_6 = (\ell_4, \ell_5)\}.\ell_6$ (that is, is $(3, 4)$ a pair of integers). This will be true if any value we can bind to ℓ_3 that satisfies its description could also be bound to ℓ_6 and satisfy its description. Since ℓ_3 is a pair and ℓ_6 is also required to be a pair, they are compatible. Furthermore, the first component of ℓ_3 is ℓ_1 and the first component of ℓ_6 is ℓ_4, so we need that any value we can bind to ℓ_1 can be bound to ℓ_4. Since ℓ_1 has to be 3, and ℓ_4 is just required to be some integer, that checks out. Similarly for ℓ_2 and ℓ_5. When any value that can be bound to a label ℓ_1 is required to also be bindable to label ℓ_2 we say that ℓ_2 is *matched* to ℓ_1. The structural matching phase is about finding a *matching*, that is, a set of such pairs, and verifying that the descriptions of the labels imply the required condition.

Things are a little more subtle though. Consider if $\{\ell_1 = 3, \ell_2 = 4, \ell_3 = (\ell_1, \ell_2)\}.\ell_3$ is contained in $\{\ell_4 = \mathsf{ints}, \ell_6 = (\ell_4, \ell_4)\}.\ell_6$ (is $(3, 4)$ a pair of the same integer). Here, ℓ_6 is matched to ℓ_3, ℓ_4 is matched to ℓ_1, and ℓ_4 is matched to ℓ_2. Everything appears to be fine. However, ℓ_4 can only be bound to a single value at a time, and so cannot be bound to both 3 and 4 at the same time. To avoid this problem, set containment only matches a label from s_2 to at most one label of s_1 or the context, but also seeks an *equating* of the labels of s_1 and the context. In this example, ℓ_6 is matched to ℓ_3, ℓ_4 could be matched to ℓ_1 and ℓ_1 equated to ℓ_2 (another possibility is to match ℓ_4 to ℓ_2 and equate ℓ_2 to ℓ_1). The descriptions of labels that are equated must imply that the two labels are always bound to the same value. Here, ℓ_1 is always 3 and ℓ_2 is always 4, so this is not the case, and set containment fails.

The declarative version of the structural matching phase asks if there exists a matching and an equating such that s_2's label is matched to s_1's label and all the matched and equated labels' descriptions satisfy the required conditions for matching and equating respectively. If two labels that are clearly always integers are matched or equated, they are simply noted for the third phase.

The third phase is to check the integer equalities noted in the structuring matching phase by generating a formula and asking a theorem prover if the formula is true. This process is mostly straightforward with a few subtleties which we do not discuss here. Essentially the descriptions of the matched/equated integer labels and those transitively referenced by them are used to make predicates which are combined into a forall/exists formula that implies the matched/equated integer labels will always be bound to the same integer. In general this formula has the form:

$$\begin{array}{l}\forall\langle\text{left integer variables}\rangle : \\ \quad \langle\text{left description predicates}\rangle \implies \\ \quad\quad \exists\langle\text{right integer variables}\rangle : \\ \quad\quad\quad \langle\text{right description predicate}\rangle \wedge \langle\text{required equalities}\rangle\end{array}$$

where the left labels and their descriptions come from the context and s_1 and the right labels and their descriptions come from s_2. This is a first-order formula in the usual theory of integer arithmetic. In general it is not in a decidable fragment of that theory for two reasons. First, we do not necessarily generate linear formulas. Second, there are existentially quantified variables. It is easy to eliminate many of these existentially quantified variables, such that the ones that remain are essential to what was written in the program. Modern SMT solvers are quite happy to try to prove formulas of the above type, but might give false negatives. Our type system is as powerful as the SMT solver used.

7. Discussion and Future Work

Overloading Contravariant functions, as well as having the contravariant domain subtyping one normally expects, also unify in interesting ways. The term $\mathsf{fn}^-(x = 3)3$ includes all functions whose domain includes 3 and that maps 3 to 3. Similarly for $\mathsf{fn}^-(x = 5)5$. The term

$$(\mathsf{fn}^-(x = 3)3) == (\mathsf{fn}^-(x = 5)5)$$

is the intersection of these two sets of functions. That is, it contains all functions whose domain includes 3 and 5, maps 3 to 3, and maps 5 to 5. That is, it is like an overloaded function doing what both these functions do. We could allow such overloading as a first-class feature, but that makes checking application and set containment much more involved. Instead we only have a second-class feature that allows more directed checking. We enforce that the domains of each function in the overload are mutually disjoint, and therefore can generate a discriminator that we can use at runtime to decide which overload applies based on the actual argument. We reject the overload if the discriminator requires function comparison. To check application, we check that the actual argument is contained in the union of the domains, and for each domain it might intersect we run the range and include it in the result.

Recursion It is easy to add a form of value recursion to the language. The runtime hardly changes, although it must account for cycles in doing some of its checks. The verifier is slightly more complicated. If recursion cannot occur in domain terms, then mostly it is a matter of accounting for cycles in the head heaps. The set containment rules were carefully designed to allow for that possibility. Value recursion where recursive uses do not occur in domain terms is probably easy to add to the set-theoretic semantics. Unfortunately, it does not seem possible in general to make this restriction.

To see this, note that types are just identity functions, and so recursive types are encoded as functions that recurse through the domain term. The runtime can handle these easily, but things are more complicated for the set-theoretic semantics and the verifier. We have some preliminary ideas about how to deal properly with recursive types. We have not worked out all the details, but suspect it requires the combination of some laziness (otherwise verifying the recursive type itself gets into an infinite loop) and possible some more cycle detection—checking containment of recursive types requires checking containment of their domains, which could recurse back to checking containment of the types. Adding recursive type constructors of higher kind could also be considered, but it's not yet clear what this would involve.

The original goal of our language was to include a more general form of recursion as in Haskell, but for a strict language. It is not too hard to devise reasonable runtime semantics for such general recursion (though there are choices to be made, especially in how recursion, effects and failure interact). However, we spent many years trying to figure out how to type check with general recursion and it was only by restricting to value recursion that we gained traction.

All quantifiers Joins express unions of values and lets provide a quantified form of union which acts like an existential quantifier. Similarly, unify expresses intersection of values, and not surprisingly there is a quantified version of intersection, called an *all quantifier*, which acts like a universal quantifier. When used with contravariant functions, all quantifiers give a kind of parameterised overloading—in other words, parametric polymorphism. For example

$$\mathsf{type}\{\mathsf{all}(t = \mathsf{types})\mathsf{fn}^-(x = \mathsf{arr}[\mathsf{nats}]i \mapsto {:}t)\mathsf{arr}[\mathsf{nats}]i \mapsto {:}t\}$$

is the type of all functions that take an array and return an array of the same type. The set-theoretic intuition for all quantifiers is quite clear, but in most cases they are not realisable. For their use with contravariant functions, there is a reasonable implementation and things seem tractable in the verifier. The most interesting part of verification is checking application of an all quantified function. It is easy to run the all quantifier domain and the function domain to form a set describing the domain, and then to check containment of the actual argument in this set. To run the range of the function we need to bind both the all quantifier parameter and the function parameter. The latter is bound to the actual argument, the question is what to do former. We use the matching inferred by the set containment to narrow the all quantifier parameter to a potentially smaller set than its domain and use that. This is like a form of type argument inference. We are still working out the details, but it looks very promising. One additional aspect is worth mentioning. Unify breaks the parametricity property that many parametrically polymorphic systems enjoy. To restore parametricity we need to add a term similar to anys that means "a value that is not allowed to be compared"; since we already have to check for function comparison, this is just an extension of that checking. Using such a term, we believe that we can build truly abstract types that enjoy parametricity.

Type classes Functions that are parameterised over type classes are sometimes thought of as functions that take an extra implicit argument, and the type checker has to infer what that extra argument should be. Scala even makes approach explicit [20]. We plan to take a similar approach to implement type classes.

Modules In many ways, modules are like records whose components can be both types and values. The types of the value components might refer to the type components, and so modules are like dependently-typed records. In fact, most module systems involve a form of limited dependent type system, usually carefully constructed to be decidable. $\lambda_{\aleph}$ has all the pieces needed to express modules: it has tuples (records are an easy extension), types as values, and dependent typing. To make an effective system, we will need to extend the verifier to properly deal with abstract types and type equalities, and to ensure that the appropriate equational properties can be decided by the verifier. We have not yet worked out the details for this, but are optimistic that it can be made to work out cleanly.

References

[1] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[2] L. Augustsson. Cayenne—a language with dependent types. In *ICFP*, pages 239–250. ACM, Sept. 1998. See also `http://www.augustsson.net/Darcs/Cayenne/html/`.

[3] H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, Apr. 1991.

[4] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, pages 105–116. ACM, 2010.

[5] E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ICFP*, pages 297–308. ACM, Sept. 2010. See also `http://www.idris-lang.org/`.

[6] L. Cardelli. A polymorphic λ-calculus with Type:Type. Technical Report SRC Research Report 10, Digital Equipment Corp., May 1986.

[7] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, pages 94–106. ACM, Sept. 2011.

[8] N. Gesbert, P. Genevès, and N. Layaïda. Parametric polymorphism and semantic subtyping: the logical connection. In *ICFP*, pages 107–116. ACM, Sept. 2011.

[9] N. Glew, T. Sweeney, and L. Petersen. Formalising the $\lambda_{\aleph}$ runtime. *ArXiv e-prints*, July 2013. URL `http://arxiv.org/abs/1307.5277`.

[10] INRIA. The Coq proof assistant. `http://coq.inria.fr/`.

[11] R. Jhala, R. Majumdat, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, volume 6806 of *LNCS*. Springer-Verlag, July 2011.

[12] L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ICFP*, pages 27–38. ACM, Sept. 2008.

[13] G. Kimmell, A. Stump, H. Eades, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV*, pages 15–25. ACM, Jan. 2012.

[14] K. Knowles, A. Tomb, J. Gronski, S. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, University of California at Santa Cruz, May 2007. URL `http://sage.soe.ucsc.edu/`.

[15] D. McAllester. *Ontic: A Knowledge Representation System for Mathematics*. MIT Press, 1989.

[16] D. McAllester. The Ontic language. Available at `http://ttic.uchicago.edu/ dmcallester/ontic-spec.ps`, June 1993.

[17] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, pages 229–240. ACM, Sept. 2008.

[18] R. P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Eindhoven University of Technology, 1973.

[19] U. Norell. Agda wiki. `http://wiki.portal.chalmers.se/agda/`.

[20] B. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360. ACM, Oct. 2010.

[21] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bharagavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278. ACM, Sept. 2011. See also `http://research.microsoft.com/en-us/projects/fstar/`.

[22] S. Weirich, J. Hsu, and R. Eisenberg. Towards dependently typed Haskell: System FC with kind equality. In *ICFP*. ACM, Sept. 2013.

[23] H. Xi. Applied type system (extended abstract). In *post-workshop proceedings of TYPES 2003*, volume 3085 of *LNCS*, pages 394–408. Springer-Verlag, 2004.

[24] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, Jan. 1999.