Chapter 1

Cardinality Analysis for Recursive Types (Draft)

Baltasar Trancón y Widemann (bt@cs.tu-berlin.de)
Markus Lepper (lepper@cs.tu-berlin.de)

Abstract: The traditional approach to recursive types in higher-order functional languages imposes strong restrictions on admissible recursion schemata: a sum of products of simple type expressions, possibly parametric, is the norm. In this paper, we advocate a more liberal type definition language, combining standard type constructors and first-order parametric polymorphism with arbitrary recursion and some useful non-free constructs. The existence of fixpoints and some other properties of the defined types, e.g., the existence of a computable equality and total order, can be inferred by abstract interpretation of the type equations in terms of *cardinality classes*.

Keywords: recursive types, polymorphism, fixpoint semantics, cardinality, abstract interpretation

1.1 INTRODUCTION

The purpose of this paper is to exemplify the static analysis of complex type definitions by means of abstract interpretation in some reasonably simple semantic domain. The motivation for such an effort is to relax some of the historically evolved restrictions of type definitions, without losing the benefits of strong static typing, both for early detection of program errors and as a guideline for efficient code generation. Instead of "outsourcing" all useful nontrivial type constructs into auxiliary libraries, we advocate the integration of semantically well-founded high-level type operations into the core of a programming language. This is already an established practice in the area of specification languages, whether intended to be executable (such as Microsoft's AsmL[6]) or not (such as Z[7]).

The paper is organized as follows: First, some desirable generalizations of common type definition techniques are outlined and their implications discussed. Then, a type definition language incorporating these generalizations is presented

in front-end syntax, back-end representation, and denotational semantics. Last, and most importantly, an abstract interpretation of type definitions is defined, enabling constructive algorithms for the relevant semantic problems that are not decidable from the set-based model alone.

1.1.1 General Type Recursion

Free data types, which essentially describe term languages, are commonly defined by systems of recursive type equations. A language might also allow other type definition, such as type aliases, parametricity or structured type definitions that do not fit into the sum-of-products pattern. These should all be orthogonal extensions of the same base mechanism, not only for uniformity, but also to benefit from each other's abstractions: E.g., combining type aliases and parametric polymorphism immediately yields type macros:

$$\begin{aligned} \mathit{binOp}[\alpha] &::= (\alpha \times \alpha) \to \alpha \\ \mathit{relation}[\alpha] &::= (\alpha \times \alpha) \to \mathit{bool} \end{aligned}$$

Expanding such definitions conveys more information to the type checker than treating them as opaque data type definitions, enabling type compatibility rules:

$$\mathit{binOp}[\alpha] = \mathit{relation}[\alpha] \iff \alpha = \mathit{bool}$$

The definitions of data types need not be restricted to the sum-of-products schema, either. Consider, e.g., this non-standard definition of nonempty α-lists:

$$\mathit{cons}[\alpha] ::= \alpha \times \big(\mathit{cons}[\alpha] + ()\big)$$

Such a definition is not commonly considered useful in an algebraic context, but it is nevertheless quite a natural thing to define in a coalgebraical or imperative context. Some fields of application require nonempty lists for modeling, e.g., the combinator $+$ of SGML and its omnipresent descendant, XML[2].

1.1.2 Non-Syntactic Type Constructors

The constructors of classical algebra (sum and product) are strongly connected to context-free syntax rules: The algebra of terms is an initial model of a classical signature. However, there are many useful type constructors whose properties cannot be handled by syntax alone. Structures such as sets and maps are slowly working their way from pure specification languages such as Z not only into auxiliary libraries, but into the core of executable languages, e.g., Microsoft's AsmL[6]. Consider the type $\mathtt{SET}\,\alpha$ of finite sets of αs. Clearly, such a type must have the following properties to be of any use:

1. The element relation $\in \subset \alpha \times \mathtt{SET}\,\alpha$ is decidable.
2. The elements of any instance of $\mathtt{SET}\,\alpha$ can be enumerated (axiom of choice).

The latter property rules out purely intensional representations. But even supposing there is an extension, the former property implies that α must provide a decidable equality. Since some other types are commonly represented intensionally, e.g., function types, the `SET` constructor turns out to be partial. Things get even more complicated when operational complexity is examined: If α not only provides an equality, but also a total order, then there are better time complexity bounds for most operations on $\texttt{SET}\,\alpha$, e.g., time complexity of $\in$ becomes $O(log\, n)$, where n is the number of elements in the set. Different representations of the `SET` type might be chosen depending on the properties of the argument type.

There are some semi-automatic solutions to these problems. The type definition language of Haskell, e.g., has the ability to derive the equality and standard order relations of a free data type (i.e., the instantiation of type classes[9] `Eq` and `Ord`), given all field types are *declared* explicitly as instances of `Eq`/`Ord`. But even the creators of Haskell admit in [3] that this mechanism is not entirely satisfactory. Given a richer type system and some cardinality information, one can do better: Equality and standard order can be inferred for all enumerable types, even for some that are conceptually intensional: If proven to be enumerable, $\texttt{POW}\,\alpha$ coincides with the extensional $\texttt{SET}\,\alpha$ and $\alpha \nrightarrow \beta$ coincides with the extensional $\alpha \nRightarrow \beta$. (See next section for the definition of these constructors)

The drawback is that, once having type constructors that require an equality relation, the equivalence properties of that relation (reflexivity, symmetry and transitivity) are part of the semantics of the type language. Then, it might not be desirable to have the user implement this relation (and guarantee its properties), as a Haskell programmer would do by instantiating `Eq` by hand. This is not a real issue, however, because a custom equality relation can be encoded as the kernel of a function, which is provably an equivalence relation: For an equivalence $eq \subseteq \alpha \times \alpha$, there is a suitable function $eq' : \alpha \to \beta$, such that $eq = \mathrm{Ker}\, eq'$, i.e.:

$$x\; eq\; y \Longleftrightarrow eq'(x) = eq'(y)$$

A custom standard order can be encoded in much the same way.

To summarize: If advanced type constructors are to be supported directly by a language, then there must be some means to check the constraints imposed on the argument types automatically, i.e., certain properties of types and also certain canonical operations must be *inferred* from the type definitions.

This approach is completely opposed to the common practice of creating runtime libraries populated with useful data types, and having semantics specified solely in terms of suggestive names and reference manuals.

1.2 DEFINING TYPES

1.2.1 Type Definition Language

We define a type definition language by the following EBNF grammar:

$$
\begin{aligned}
T ::=\ & V \left[\underline{[}\ T\ \underline{]}\right] \mid \underline{\{\}} \mid \underline{()} \\
& \mid T \underline{+} T \mid T \underline{\times} T \mid T \underline{\to} T \mid T \underline{\not\to} T \mid T \underline{\not\Rightarrow} T \\
& \mid \underline{\texttt{OPT}}\, T \mid \underline{\texttt{SEQ}}\, T \mid \underline{\texttt{SET}}\, T \mid \underline{\texttt{BAG}}\, T \mid \underline{\texttt{POW}}\, T \\
V ::=\ & \alpha \mid \beta \mid \cdots \mid \tau \mid \ldots \\
E ::=\ & V \left[\underline{[}\ V\ \underline{]}\right] \underline{::=} T
\end{aligned}
$$

Types are defined by systems of arbitrarily recursive and (potentially) polymorphic type equations. Type equations E bind type variables V, optionally with a polymorphic parameter, to type expressions T. The language of type expressions is induced by:

1. type variables (instantiated with types if parametric)
2. the empty type $\{\}$ and the void type $()$
3. Cartesian product $(\times)$ and the disjoint sum $(+)$ operators
4. spaces of total $(\to)$ and partial $(\not\to)$ functions and finite maps $(\not\Rightarrow)$
5. collection monads of optionals (`OPT`), finite subsets (`SET`), finite multisets (`BAG`), finite sequences (`SEQ`) and arbitrary subsets (`POW`).

For the sake of simplicity and limited space in this paper, we consider only parametric types (constructors) with at most one polymorphic parameter. Since there are no tuples of types, there is no way to simulate n polymorphic parameters as of now, but the system can be easily generalized.

1.2.2 Type Representation Language

The front-end language, modeled after commonly used notations, needs to be slightly transformed into a back-end representation to accomodate semantic reasoning:

$$
\begin{aligned}
T' ::=\ & V \mid F \left[\underline{[}\ T'\ \underline{]}\right] \mid \underline{\{\}} \mid \underline{()} \\
& \mid T' \underline{+} T' \mid T' \underline{\times} T' \mid T' \underline{\to} T' \mid T' \underline{\not\to} T' \mid T' \underline{\not\Rightarrow} T' \\
& \mid \underline{\texttt{OPT}}\, T' \mid \underline{\texttt{SEQ}}\, T' \mid \underline{\texttt{SET}}\, T' \mid \underline{\texttt{BAG}}\, T' \mid \underline{\texttt{POW}}\, T' \\
F ::=\ & W \mid [\underline{\lambda}\ V\ \underline{.}]\ T' \mid W\ \underline{@}\ R \\
W ::=\ & \phi \mid \phi_1 \mid \ldots \\
R ::=\ & \underline{\mu}\ (W\ \underline{::=}\ F)^+
\end{aligned}
$$

In the type representation language, the recursion is made explicit. Recursion schemata (recursions for short) R are first order citizens, with the μ-operator binding constructor variables to defining equations and yielding the least fixpoint of the given mutually recursive constructor family transformation. Constructors are made up of type λ-abstractions and constructor references. Recursive constructor references are μ-bound constructor variables. A constructor reference ϕ that is *not* recursive is expressed as the projection ϕ @ r from a recursion r. Types, finally, are built with the standard operators and explicit constructor instantiation.

The μ operator comes with the builtin fixpoint semantics of free data types (initial algebras). There are other fixpoint semantics, e.g. cofree data types (final coalgebras). However, the denotational semantics and the cardinality analysis of types defined with such operators are outside the scope of this paper.

To summarize, there is a twofold abstraction involved in the definition of types:

$$\begin{array}{ccccc} \textit{types} & & \textit{constructors} & & \textit{recursions} \\ \textit{units of discourse} & \longrightarrow & \textit{units of semantics} & \longrightarrow & \textit{units of declaration} \end{array}$$

To simplify the notation of non-polymorphic types, we shall use a type t as a constant constructor, with the meaning $\lambda\tau \,.\, t$, where τ does not occur freely in t. Dually, we shall use a constant constructor f as a type, with the meaning $f[t]$, where t is an arbitrary well-formed type expression, e.g. $()$. Then, constructor semantics can be defined uniformly for all type equations.

Examples

The mapping from front-end to back-end language is so straightforward that we do not define it formally in this paper. Essentially, groups of mutually recursive equations are prefixed with a μ operator, and all other type references are resolved and replaced by explicit @ expressions.

Some simple type definitions, both in front-end and back-end notation:

1. The natural numbers *Nat*:

$$nat ::= nat + () \qquad \longmapsto \qquad \mu\, nat ::= nat + ()$$

2. The homogenous lists *HList*:

$$hlist[\alpha] ::= \big(\alpha \times hlist[\alpha]\big) + () \qquad \longmapsto \qquad \mu\, hlist ::= \lambda\alpha \,.\, \big(\alpha \times hlist[\alpha]\big) + ()$$

3. The pure lisp lists *LList* defined in terms of homogenous lists. Note the explicit name resolution in the back-end:

$$llist ::= hlist[llist] \qquad \longmapsto \qquad \mu\, llist ::= (hlist \mathbin{@} HList)[llist]$$

4. The homogenous lists again, defined in terms of (possibly) empty and nonempty lists:

$$\begin{array}{r}opt[\alpha] ::= cons[\alpha] + () \\ cons[\alpha] ::= \alpha \times opt[\alpha]\end{array} \quad \longmapsto \quad \mu \left\{ \begin{array}{l} opt ::= \lambda\alpha\,.\,cons[\alpha] + (), \\ cons ::= \lambda\alpha\,.\,\alpha \times opt[\alpha] \end{array} \right.$$

1.2.3 Denotational Semantics

We shall give naïve partial denotational semantics based on plain set-theory for types ($[\![\ ]\!]_{\mathrm{T}}$), constructors ($[\![\ ]\!]_{\mathrm{F}}$)and recursions ($[\![\ ]\!]_{\mathrm{R}}$), respectively. The semantic domains are:

1. the objects of the category $\underline{Set}$ for types
2. the partial endofunctors $\underline{Set} \nrightarrow \underline{Set}$ for constructors
3. the finite W-indexed families of functors $W \nRightarrow \underline{Set} \nrightarrow \underline{Set}$ for recursions

Note that we will regard only the object-mapping part of functors in this paper.

A context Γ denotes a pair of valuation maps $\Gamma_{\mathrm{T}} + \Gamma_{\mathrm{F}}$, such that $\Gamma_{\mathrm{T}} : V \nrightarrow \underline{Set}$ and $\Gamma_{\mathrm{F}} : W \nrightarrow (\underline{Set} \nrightarrow \underline{Set})$. See table 1.1 for the semantic equations.

An object (type/constructor/recursion) is said to be well-formed, iff its denotational semantics are defined. There are many ill-formed objects: Some of these contain merely syntactical errors, e.g., references to unbound variables. A more interesting class of ill-formed objects are those involving a non-existent fixpoint construction. The following example has been known to be ill-formed since the time of Cantor:

$$\mu s ::= \mathtt{POW}\, s$$

The following definition has no semantics either. Both the assumption that s is empty and that s is nonempty lead to a contradiction:

$$\mu s ::= s \to \{\}$$

In any case, the semantic domain of general sets is far too powerful and thus of no use in automatic verification of type properties. Since the solution space for recursions are the partial set endofunctors, it is even undecidable whether a candidate is actually the fixpoint of a given functor transformation. In the next section, we will present an extremely simplified semantic domain for the abstract interpretation of type definitions. Though reducing the complexity of the system enough to obtain computable fixpoints, our model preserves many interesting properties of the defined types.

1.3 CARDINALITY CLASSES

Type definitions fall into one of six cardinality classes, all well-known from standard set theory:

$$C = \{\mathbf{0} < \mathbf{1} < n < \omega < \mathfrak{C} < \lightning\}$$

See table 1.2 for the pronounciation of these symbols. Each cardinality class associates some properties with its member types:

1. A type of cardinality $\mathbf{0}$ has no instance.
2. A type of cardinality $\mathbf{1}$ has exactly one instance.
3. A type of cardinality n or greater has more than one instance.
4. A type of cardinality n or less has finitely many instances.
5. A type of cardinality ω or greater has infinitely many instances.
6. A type of cardinality ω or less has a finite extensional representation.
7. A type of cardinality ω or less is recursively enumerable.
8. A type of cardinality $\mathfrak{C}$ has more than countably many instances.
9. A type of cardinality $\lightning$ is ill-defined.

These properties are not of purely theoretical interest, but have some important practical corollaries:

1. A type of cardinality $\mathbf{0}$ or $\mathbf{1}$ needs no representation in memory at all.
2. A type of cardinality n can be encoded in a fixed number of bits in memory.
3. A type of cardinality ω must be encoded using extensible bit vectors or pointers.
4. A type of cardinality ω or less induces a computable total order.
5. A type of cardinality $\mathfrak{C}$ must be encoded intensionally.
6. A type of cardinality $\lightning$ is a programmer's error.
7. A type of cardinality $\mathbf{0}$ or $\mathbf{1}$ with a nontrivial definition is probably a programmer's error and should trigger a warning. Consider, e.g., the following erroneous definition of homogenous lists that has cardinality $\mathbf{0}$:

$$\mathit{hlist}[\alpha] ::= \big(\alpha \times \mathit{hlist}[\alpha]\big) \times ()$$

The abstract interpretation of type definitions in terms of cardinality classes is defined as follows:

1. The type constants $\{\}$ and $()$ are mapped to cardinalities $\mathbf{0}$ and $\mathbf{1}$, respectively.

2. The interpretations of the builtin constructors are shown in tables 1.3–1.5.

Since the space of cardinality functions $C \to C$ is finite, it is decidable whether a finite family of cardinality functions is in fact a fixpoint of a given transformation (the abstract interpretation of a recursion). In the remaining sections of this paper, we shall go further and actually give a computable construction for cardinality fixpoints.

1.3.1 Monotonicity

In order to compute least fixpoints of recursive cardinality equations, we shall use a straightforward bottom-up approximation based on [8]. The domain of cardinality classes C, our interpretation of closed type expressions, is trivially a CPO $(C, \leq)$ with the bottom element $\mathbf{0}$. Furthermore, the top element $\lightning$ denotes ill-formedness.

The domain of unary cardinality functions, our interpretation of constructors, can be ordered pointwise to obtain a CPO $(C \to C, \sqsubseteq)$ with bottom element $\mathit{const}\,\mathbf{0}$:

$$f \sqsubseteq g \iff \forall c \,.\, f(c) \leq g(c)$$

The last step of abstraction is the interpretation of recursions as monotonic transformations of (families of) cardinality functions. Since monotonicity is most naturally formulated for unary functions, the binary standard operations are treated as families of *sections*, i.e., partially instantiated in either their left or right argument with any possible value. Each section of a standard operation corresponds to a row or column of tables 1.3/1.4. Sections are denoted in parentheses with the uninstantiated argument set to a placeholder _.

It is easy to show (using the given tables) that repeated application of most of the given unary cardinality operations f is monotonic, i.e., $x \leq f(x)$. There are two kinds of exceptions:

1. The constant operations $(\mathbf{0} \times _)$, $(_ \times \mathbf{0})$, $(\mathbf{0} \to _)$, $(_ \to \mathbf{1})$, $(\mathbf{0} \nrightarrow _)$, $(_ \nrightarrow \mathbf{0})$, $(\mathbf{0} \nRightarrow _)$, and $(_ \nRightarrow \mathbf{0})$. For these, however, the weak monotonicity $f(x) \leq f\big(f(x)\big)$ holds.
2. The alternating operation $(_ \to \mathbf{0})$. A recursion involving this operation and none of the constant operations has no fixpoint. Thus, if such a circle is detected, the fixpoint approximation can be aborted (i.e., set to $\lightning$), instead of alternating infinitely between $\mathbf{0}$ and $\mathbf{1}$.

1.3.2 Totality

Note that all fully monotonic operations are necessarily strict with respect to the top element: $f(\lightning) = \lightning$. Thus the interpretation of $\lightning$ as the ill-defined cardinality is justified. As a consequence, even though the calculus of recursive type definitions has partial semantics, the abstract interpretation in terms of cardinality is total with a one-to-one correspondence between ill-defined set semantics and cardinality class $\lightning$.

1.3.3 Classes of cardinality operations

The subranges $C_c = \{\mathbf{0}, \dots, c\}$ of C are closed under certain classes O_c of operations. An operation f shall be called *c-faithful*, iff C_c is closed under f:

$$x \leq c \Longrightarrow f(x) \leq c$$

1. All standard cardinality operations except for sections instantiated with $\lightning$ and the exponential closure operations `SET`, `BAG` and `POW` are $\mathfrak{C}$-faithful.
2. The only $\mathfrak{C}$-faithful operations that are not ω-faithful are the sections instantiated with $\mathfrak{C}$ and the exponential operations $(_ \rightarrow \omega)$, $(_ \nrightarrow \omega)$ (but not $(_ \nRightarrow \omega)$), and `POW` (but not `SET`).
3. The only ω-faithful operations that are not n-faithful are the sections instantiated with ω and the closure operations `BAG` and `SEQ`.
4. The n-faithful operations are also called *polynomial*.

1.3.4 Short-Cut Recursion

Obviously, abstract interpretation in terms of cardinality classes *alone* is not sufficient to deal with recursion. Even the most basic recursive examples exceed the power of the cardinality calculus presented so far. Consider the recursive definition of natural numbers:

$$\mu\, nat ::= nat + ()$$

If the least fixpoint is approximated naïvely, the cardinality of *nat* will be approximated by $\mathbf{0} \rightarrow \mathbf{1} \rightarrow n \rightarrow n \rightarrow \dots$, and never exceed n. Since we intend to compute a cardinality fixpoint in finitely many approximation steps, but the set-theoretic fixpoint takes ω steps to approximate, the naïve approach must fail.

In order to solve this problem, we need means to detect recursive paths in the type expressions, i.e., paths that are going to repeat ω times in the fixpoint, and perform all ω steps *at once* in the cardinality calculus. Thus, a purely algebraic notion of type expression terms is ruled out. Instead, we shall formulate a recursion-capable cardinality inference algorithm based on a coalgebraic notion of *type expression graphs*.

Informally, every subexpression in a system of type definitions is assumed to have a unique node identity $i \in \mathbb{N}$. Subexpression nodes are labeled with their root operation. The number of outgoing edges at each node shall correspond with the number of arguments of the operation. Furthermore, there shall be a total order among the edges originating from a single node, reflecting the textual order of the arguments. A graph representation of the solution for an equation is easily obtained by mapping the left hand side node onto the root of the right hand side (see figure 1.1).

Now we can refine the abstract interpretation by annotating each computed cardinality with the set of expression identities that *contribute monotonically*:

1. The contributor set of a constant operation is empty.

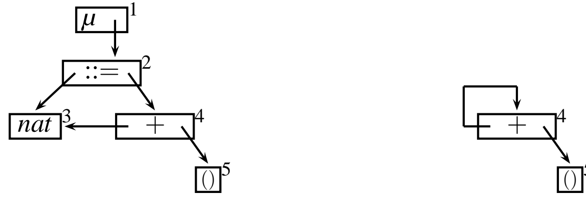

FIGURE 1.1. The equation and solution graphs of *nat*

2. The contributor set of a monotonic operation is the union of the root node identity and the contributor set of the arguments (including the fixed argument of a section). The pathological operation $(_ \to \mathbf{0})$ can be treated like a monotonic operation in the recursive case, yielding ↯.

Whenever the identity of an operation node is found in the contributor set of one of its arguments, there is a recursive subexpression that will be unrolled ω times in the least fixpoint of the type definition. This recursion *must* be monotonic: If there was a non-monotonic operation involved, then the contributor set would have been cleared at that point. Whenever a monotonic recursion is detected, a different interpretation of the type operation is employed, reflecting the cardinality effect of ω successive applications of the operation.

Table 1.6 shows the operations that behave differently in the recursive case. The meta-variable r stands for a recursive subexpression, i.e., a subexpression whose contributor set contains the identity of the root node. Operations not occurring in the table need no special treatment in the recursive case.

Illustrated graphically, the solution is approximated by bottom-up construction of trees that can be mapped homomorphically onto the circular solution graph. The recursion handling mechanism takes place whenever the mapping is found to be non-injective. Then, the two identified nodes, being beginning and end of a circle, are collapsed and the resulting single node is updated to reflect the cardinality of ω iterations of that circle (see figure 1.2).

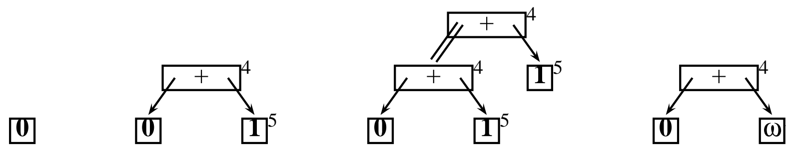

FIGURE 1.2. Fixpoint approximation for *nat*

1.4 CONCLUSION

We have presented a type definition language that aims at incorporating some state-of-the-art and highly desirable generalizations into the standard model of parametric free data types, namely general and mutual recursion and some advanced type constructors. We have shown that cardinality analysis is the key to several important properties of the resulting type definitions, including well-formedness, and how to accomplish terminating cardinality inference by abstract interpretation in a recursion-aware domain of cardinality classes.

1.4.1 Future Work

Formal Proof The inferred cardinality results have been validated in a prototypic implementation, but a formal proof of our method has yet to be done. The topic is still quite new, and we hope to gain insights from future extensions and generalizations that will help us in formulating a concise proof.

Refinement of Interpretation It is easy to verify that all of the six cardinality classes described in this paper are indispensable for consistent interpretation of type constructors. Introducing new classes above $\mathfrak{C}$, though of interest to set theory, is pointless for (constructive) data types. Refining the abstract interpretation domain beyond cardinality might induce classes slightly above or below ω with different properties, however. This should be a future topic of research, especially the range of countable but *not enumerable* types obtained with non-convex constructors, e.g., general Chomsky grammars or semantic subtypes.

Cofree Semantics The operator μ and its least fixpoint semantics is not the only way to give semantics to recursive type definitions. There might also be an operator ν with geatest fixpoint semantics, yielding final coalgebra models of types. Since final coalgebras tend to behave badly in terms of cardinality, e.g. $\nu t ::= t + t$ is already continuous, the more interesting fixpoint operator might be $\bar{\mu}$, yielding a submodel of the final coalgebra that contains only instances with a finite (albeit cyclical) graph representation, a very powerful and natural model of semantic nets and object-oriented systems. It is not yet known whether our abstract interpretation approach can be generalized to support such fixpoint computations at all, and if it does, which refinements are necessary.

Cardinality and Functorial Semantics The possibility of generalization of type constructors to full functors (i.e., with a morphism-mapping part) has been a topic of active research in recent years, see e.g. [5]: Most of the conceivable type constructors have a generalization to morphisms, allowing the automatic lifting of operations to composite types. It is not yet known whether cardinality analysis might also provide a clue to properties of the morphism construction associated with a type construction.

Further Uses of Cardinality Info There are many more applications for cardinality information. E.g., a type of class ω or less induces a canonical pretty printing function. A less trivial application is a special treatment of types of class **1**: By collapsing all of these types into a single canonical void type $()$, and by adding a type equation $\alpha = () \to \alpha$, it is possible to eliminate the distinction between values and functions, and thus between application and composition, reconciling calulations based on λ-terms and morphisms. This topic is currently under the authors' investigation.

1.4.2 Related Work

There are many theoretical results and areas of application for abstract interpretation of *programs*. See [4] for a comprehensive treatise. To our knowledge, there are few examples of abstract interpretation of a type language. However, we expect this to change in the future, because type languages are becoming increasingly powerful and actually competing with computational languages in complexity.

The only related application of cardinality analysis we are aware of is the static analysis of solution spaces for Prolog queries [1]. This work is about actual computations and thus about finite cardinalities, whereas we have a statical and transfinite approach.

ACKNOWLEDGEMENTS

Thanks to Jacob Wieland and the ÜBB compiler construction group at *Technische Universität Berlin* for inspiring discussions.

REFERENCES

[1] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, pages 457–471. The MIT Press, 1994.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, `http://www.w3.org/TR/2000/REC-xml`.

[3] R. Hinze and S. Jones. Derivable type classes, 2000.

[4] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, October 91.

[5] C. B. Jay, G. Belle, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[6] microsoft research/FSE. *AsmL 1.5 online documentation* `http://www.research.microsoft.com/fse/asml/doc`.

[7] J. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice Hall International, 2 edition, 1992.

[8] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[9] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

$$
\begin{aligned}
[\![\Gamma \vdash _]\!]_{\mathrm{T}} &: T' \nrightarrow \underline{Set} \\
[\![\Gamma \vdash \tau]\!]_{\mathrm{T}} &= \Gamma_{\mathrm{T}}(\tau) \\
[\![\Gamma \vdash f[t]]\!]_{\mathrm{T}} &= [\![\Gamma \vdash f]\!]_{\mathrm{F}}\big([\![\Gamma \vdash t]\!]_{\mathrm{T}}\big) \\
[\![\Gamma \vdash \{\}]\!]_{\mathrm{T}} &= \emptyset \\
[\![\Gamma \vdash ()]\!]_{\mathrm{T}} &= \{\star\} \\
[\![\Gamma \vdash t \times u]\!]_{\mathrm{T}} &= [\![\Gamma \vdash t]\!]_{\mathrm{T}} \times [\![\Gamma \vdash u]\!]_{\mathrm{T}} \\
[\![\Gamma \vdash t + u]\!]_{\mathrm{T}} &= [\![\Gamma \vdash t]\!]_{\mathrm{T}} + [\![\Gamma \vdash u]\!]_{\mathrm{T}} \\
[\![\Gamma \vdash t \to u]\!]_{\mathrm{T}} &= [\![\Gamma \vdash u]\!]_{\mathrm{T}}^{[\![\Gamma \vdash t]\!]_{\mathrm{T}}} \\
[\![\Gamma \vdash t \nrightarrow u]\!]_{\mathrm{T}} &= \bigcup_{X \in [\![\Gamma \vdash \mathtt{POW}\, t]\!]_{\mathrm{T}}} [\![\Gamma \vdash u]\!]_{\mathrm{T}}^{X} \\
[\![\Gamma \vdash t \not\Rightarrow u]\!]_{\mathrm{T}} &= \bigcup_{X \in [\![\Gamma \vdash \mathtt{SET}\, t]\!]_{\mathrm{T}}} [\![\Gamma \vdash u]\!]_{\mathrm{T}}^{X} \\
[\![\Gamma \vdash \mathtt{OPT}\, t]\!]_{\mathrm{T}} &= [\![\Gamma \vdash t]\!]_{\mathrm{T}} \cup \{-\} \\
[\![\Gamma \vdash \mathtt{SET}\, t]\!]_{\mathrm{T}} &= \big\{X \subseteq [\![\Gamma \vdash t]\!]_{\mathrm{T}} \mid X \text{ is finite}\big\} \\
[\![\Gamma \vdash \mathtt{BAG}\, t]\!]_{\mathrm{T}} &= \bigcup_{X \in [\![\Gamma \vdash \mathtt{SET}\, t]\!]_{\mathrm{T}}} \mathbb{N}_{+}{}^{X} \\
[\![\Gamma \vdash \mathtt{SEQ}\, t]\!]_{\mathrm{T}} &= \bigcup_{k \in \mathbb{N}} [\![\Gamma \vdash t]\!]_{\mathrm{T}}^{k} \\
[\![\Gamma \vdash \mathtt{POW}\, t]\!]_{\mathrm{T}} &= \big\{X \subseteq [\![\Gamma \vdash t]\!]_{\mathrm{T}}\big\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash _]\!]_{\mathrm{F}} &: F \nrightarrow \underline{Set} \nrightarrow \underline{Set} \\
[\![\Gamma \vdash \phi]\!]_{\mathrm{F}} &= \Gamma_{\mathrm{F}}(\phi) \\
[\![\Gamma \vdash \lambda \tau \,.\, t]\!]_{\mathrm{F}}(u) &= [\![\Gamma \oplus \{\tau \mapsto u\} \vdash t]\!]_{\mathrm{T}} \\
[\![\Gamma \vdash \phi \,@\, r]\!]_{\mathrm{F}} &= [\![\Gamma \vdash r]\!]_{\mathrm{R}}(\phi)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash _]\!]_{\mathrm{R}} &: R \nrightarrow W \not\Rightarrow \underline{Set} \nrightarrow \underline{Set} \\
[\![\Gamma \vdash \mu(\phi_i ::= f_i)]\!]_{\mathrm{R}} &= S_0
\end{aligned}
$$

where S_0 is the least fixpoint of the functor family transformation Φ, defined as:

$$\Phi(S)(\phi_i) = [\![\Gamma \oplus S \vdash f_i]\!]_{\mathrm{F}}$$

TABLE 1.1. Denotational Semantics

0	*empty*
1	*void*
n	*finite*
ω	*discrete*
$\mathfrak{C}$	*continuous*
$\lightning$	*unstable*

TABLE 1.2. Cardinality classes

+	**0**	**1**	n	ω	$\mathfrak{C}$	↯
0	**0**	**1**	n	ω	$\mathfrak{C}$	↯
1	**1**	n	n	ω	$\mathfrak{C}$	↯
n	n	n	n	ω	$\mathfrak{C}$	↯
ω	ω	ω	ω	ω	$\mathfrak{C}$	↯
$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
↯	↯	↯	↯	↯	↯	↯

×	**0**	**1**	n	ω	$\mathfrak{C}$	↯
0	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	n	ω	$\mathfrak{C}$	↯
n	**0**	n	n	ω	$\mathfrak{C}$	↯
ω	**0**	ω	ω	ω	$\mathfrak{C}$	↯
$\mathfrak{C}$	**0**	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
↯	**0**	↯	↯	↯	↯	↯

TABLE 1.3. Polynomial binary constructors

$\rightarrow$	**0**	**1**	n	ω	$\mathfrak{C}$	↯
0	**1**	**1**	**1**	**1**	**1**	**1**
1	**0**	**1**	n	ω	$\mathfrak{C}$	↯
n	**0**	**1**	n	ω	$\mathfrak{C}$	↯
ω	**0**	**1**	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
$\mathfrak{C}$	**0**	**1**	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
↯	**0**	**1**	↯	↯	↯	↯

$\not\twoheadrightarrow$	**0**	**1**	n	ω	$\mathfrak{C}$	↯
0	**1**	**1**	**1**	**1**	**1**	**1**
1	**1**	n	n	ω	$\mathfrak{C}$	↯
n	**1**	n	n	ω	$\mathfrak{C}$	↯
ω	**1**	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
$\mathfrak{C}$	**1**	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯
↯	**1**	↯	↯	↯	↯	↯

$\nRightarrow$	**0**	**1**	n	ω	$\mathfrak{C}$	↯
0	**1**	**1**	**1**	**1**	**1**	**1**
1	**1**	n	n	ω	$\mathfrak{C}$	↯
n	**1**	n	n	ω	$\mathfrak{C}$	↯
ω	**1**	ω	ω	ω	$\mathfrak{C}$	↯
$\mathfrak{C}$	**1**	↯	↯	↯	↯	↯
↯	**1**	↯	↯	↯	↯	↯

TABLE 1.4. Exponential binary constructors

	OPT	SEQ	SET	BAG	POW
0	**1**	**1**	**1**	**1**	**1**
1	n	ω	n	ω	n
n	n	ω	n	ω	n
ω	ω	ω	ω	ω	$\mathfrak{C}$
$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯	↯	↯
↯	↯	↯	↯	↯	↯

TABLE 1.5. Monadic constructors

	$(r \rightarrow _)$	$(r \not\twoheadrightarrow _)$	$(_ \not\twoheadrightarrow r)$	$(r \nRightarrow _)$ $(_ \nRightarrow r)$	$(_ + r)$ $(r + _)$	OPT SEQ SET BAG	POW
0	↯	**1**	**1**	**1**	**0**	ω	↯
1	**1**	↯	ω	ω	ω	ω	↯
n	↯	↯	ω	ω	ω	ω	↯
ω	↯	↯	$\mathfrak{C}$	ω	ω	ω	↯
$\mathfrak{C}$	↯	↯	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	↯	↯
↯	↯	↯	↯	↯	↯	↯	↯

TABLE 1.6. Recursive operations