Depth Exam: Observations and Comments

Andrei Lapets

December 20, 2007

Contents

1 "What do Types Mean?" [Rey]

The author reviews terminology for two distinct ways of assigning semantics to a typed language, and provides an example of each for a simple version of a call-by-name lambda calculus. The intrinsic semantics for the typed language is defined by recursively mapping typing derivations to meanings. Under this semantics, the derivations are mapped to one of a collection of distinct mathematical domains, one for each distinct type. The untyped semantics is defined recursively over the syntactic form of an expression, and all expressions are mapped into a single, universal domain into which both primitive values and functions over that domain can be embedded.

Next, the authors relate the family of domains used in the intrinsic definition with the universal domain used in the untyped definition. This is done by constructing a family of relations indexed by types, one for each intrinsic domain. To deal with recursion, the relations are shown to be strict and chain-complete, and then this is used to show that, for the same expressions, the meanings produced by the two semantic mappings are related by this family of relations. The authors then relate a type-indexed family of embedding-retraction function pairs to the family of relations, and use this to relate the meaning of a derivation (the intrinsic semantics) to the untyped semantics. Since the untyped semantics depends only on the conclusion of the derivation, this shows that the intrinsic semantics is in fact independent of the derivation itself, which makes the intrinsic semantics coherent, as different proofs leading to the same conclusion must have the same meaning.

Finally, the authors show that each type corresponds to not just a subset over the universal domain, but a partial equivalence relation defined over that subset under which two values $y, y' \in U$ are equivalent if they represent the same value of the type corresponding to that subset. The untyped semantics, when coupled with the partial equivalence relations, is then the extrinsic semantics.

One major observation about the example language used in this work is that its typing rules appear to be syntax-directed (for every conclusion, there exists exactly one derivation), making the language trivially coherent. Thus, it is interesting to see whether the technique used to prove coherence can actually be extended to a language in which multiple derivations can have the same judgement. We consider the trivial example language:

$$\begin{array}{rrcl} \text{expressions} & p & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots \mid p' + p'' \\ \text{types} & \tau & ::= & \textbf{nat} \mid \textbf{int} \end{array}$$

and its type system.

$$\textsc{Nat}\ \frac{n \in \{0,1,2,3,\ldots\}}{\vdash n : \textbf{nat}} \qquad \textsc{Int}\ \frac{n \in \{-1,-2,-3,\ldots\}}{\vdash n : \textbf{int}} \qquad \textsc{NatToInt}\ \frac{\vdash p : \textbf{nat}}{\vdash p : \textbf{int}}$$

$$\textsc{Plus}\ \frac{\vdash p' : \tau \quad \vdash p'' : \tau}{\vdash p' + p'' : \tau}$$

This language allows us to construct a derivation concluding that $\vdash 1+2 : \mathbf{int}$ in two different ways.

$$\dfrac{\dfrac{\dfrac{}{\vdash 1 : \mathbf{nat}}}{\vdash 1 : \mathbf{int}} \quad \dfrac{\dfrac{}{\vdash 2 : \mathbf{nat}}}{\vdash 2 : \mathbf{int}}}{\vdash 1 + 2 : \mathbf{int}} \qquad \dfrac{\dfrac{}{\vdash 1 : \mathbf{nat}} \quad \dfrac{}{\vdash 2 : \mathbf{nat}}}{\dfrac{\vdash 1 + 2 : \mathbf{nat}}{\vdash 1 + 2 : \mathbf{int}}}$$

If we used an approach similar to that of the author in defining an intrinsic semantics for this language, we might assume that $[\![\mathbf{nat}]\!] \subset [\![\mathbf{int}]\!]$ and that $+$ is mathematical addition of integers over the domain $[\![\mathbf{int}]\!]$. Then, the case for (Plus) might be defined as follows.

$$[\![\dfrac{\mathcal{P}(\vdash p' : \tau) \quad \mathcal{P}(\vdash p'' : \tau)}{\vdash p' + p'' : \tau}]\!] = [\![\mathcal{P}(\vdash p' : \tau)]\!] + [\![\mathcal{P}(\vdash p'' : \tau)]\!]$$

It appears that the author's method for proving coherence does extend to this case, at least in this oversimplified language. If we defined an untyped semantics by providing an embedding-retraction pair for integers, it would be easy to show that it is also an embedding-retraction pair for every subset of the integers, including **nat**. Thus, we would find that the two type-indexed relations between intrinsic meanings and the universal domain would be such that $\rho[\mathbf{nat}] \subset \rho[\mathbf{int}]$. It should be possible to then establish an equivalence between the intrinsic and untyped semantics of an expression in the same manner, with the addition of a subtyping rule which would look something like:

$$[\![\dfrac{\mathcal{P}(\vdash p : \mathbf{nat})}{\vdash p : \mathbf{int}}]\!] = [\![\mathcal{P}(\vdash p : \mathbf{nat})]\!] = \psi_{\mathbf{nat}}[\![p]\!] = \psi_{\mathbf{int}}[\![p]\!],$$

where our induction hypothesis is that $[\![\mathcal{P}(\vdash p : \mathbf{nat})]\!] = \psi_{\mathbf{nat}}[\![p]\!]$ and assuming we have established that $\forall p : \mathbf{nat}$, $\psi_{\mathbf{nat}}\phi_{\mathbf{nat}}p = \psi_{\mathbf{int}}\phi_{\mathbf{int}}p$ (as $[\![\mathbf{nat}]\!] \subset [\![\mathbf{int}]\!]$, the existence of this fact is not counterintuitive).

One interesting result to obtain is to specify exactly the conditions under which a language and its intrinsic semantics can be proven coherent in this manner (i.e. by specifying an untyped semantics and using the methodology in this paper to relate the two meanings of an expression). Ideally, one would like this to be true under any conditions, but that might not be the case. The author mentions that it should be possible to extend this technique to a polymorphic language, though it does not seem any further work was done in this direction, and the author's attempts to extend the technique to a language with intersection types was not successful.

It is also worth noting that extending the untyped semantics to a language with multiple primitive types would require specifying separate embedding-retraction pairs for each base type. This seems reasonable, however, as they would only be used in the base cases of the definition of the untyped semantics.

Another minor observation is that it is not made clear why the languages for which this methodology can work must be call-by-name.

2 "Nested Datatypes" [BM98]

One way to view the semantics of regular datatypes is to treat them as sets which are fixed points of the functors. The authors try to lift this semantics to one particular generalization of regular datatypes, "nested" datatypes. These are distinguished from regular polymorphic datatypes (which are essentially monomorphic datatypes parameterized by a type) in that a different type parameter can be supplied to the type constructor on the right-hand side of the datatype definition.

After providing a somewhat unconvincing example program which uses nested datatypes, the authors review initial algebra semantics for regular datatypes. They then try to lift this approach by rewriting a nested datatype in a point-free form using composition, product, and sum lifted to *functors* (as opposed to sets), rewrite the recursion equation in open form as a natural transformation on functors, and define the nested datatype to be the fixed point of this natural transformation. Unfortunately, there are some problems with this approach. Defining a fold operation over such a type results in a function that has a rank-2 type signature. It also means that the fold for regular datatypes is not an instance of this fold, and that trying to define the usual list datatype using this approach means performing some extra, inelegant functor manipulations.

Datatypes are arguably a relatively simple programming feature, and if there is any benefit to using some generalized form of datatypes instead of simply introducing a much more powerful language (such as one with dependent types), it is that even when extended in some way, datatypes will remain for programmers a simple and accessible way to represent data. While nested datatypes are more expressive, none of the interpretations presented in this work help gain an intuitive understanding of exactly how they are more expressive (which, in my opinion, should be one goal for a semantics of datatypes). Overall, it is not convincing at all that fixed points of natural transformations are a reasonable approach to the problem.

This leads to another issue, the question of how exactly generalizations of datatypes can be used. It is difficult to find good examples of uses for nested datatypes, and one major limitation of this particular generalization is that the return types of the constructors cannot be specified. For example, we might consider a datatype for an abstract syntax of expressions in which the parameter of the datatype is the type of the expression.

```
datatype Exp a = N    :: Int -> Exp Int
               | B    :: Bool -> Exp Bool
               | If   :: Exp Bool -> Exp a -> Exp a -> Exp a
               | Pair :: Exp a -> Exp b -> Exp (a, b)
               | Fst  :: Exp (a, b) -> Exp a
               | Snd  :: Exp (a, b) -> Exp b
```

Given any function of type `Exp a -> Exp a` (which might be a program transformation or optimization), we can be certain that the function preserves the type of the abstract syntax

expression represented using this datatype. Examples like this also suggest an alternate semantics for nested datatypes. Particularly, it might be worth trying to interpret such datatypes as equalizers of fixed points of functors (type parameters are not just a form of polymorphism, but also a way to encode properties of or restrictions on the data being represented). It is worth investigating what the type of a fold operation might be for this variation on datatypes.

Generally, a distinction should be made between the use of datatype parameters which are truly types, and datatype parameters which are meant simply to restrict the algebra and preserve invariants within the instances of a datatype. The only time a type such as `eval: Exp a -> a` might be useful is when performing a meta-evaluation in which the semantics of the language is embedded. [1].

3 "Some Practical Aspects of Dependent Datatypes" [Xi99]

In this paper, the author directs our attention to a potential disparity between the way type-checking is performed on pattern matching expressions in a typed functional language, and the sequential strategy used to evaluate those pattern-matching expressions in that language. The motivation provided involves type checking in a dependently typed setting (by ignoring the sequential evaluation strategy, the types inferred for pattern arguments are not always as refined as they can be). However, the resulting technique and algorithm apply to any functional language with pattern matching, and are useful for checking whether a set of patterns is exhaustive.

The strategy used to resolve the problem is to transform a collection of potentially overlapping patterns $p_1, \ldots, p_n$ into a potentially larger collection of patterns $p'_1, \ldots, p'_m$ which are mutually disjoint. This involves computing for each p_i the union of complements for the other patterns $\{p_j\}_{j \neq i}$. Essentially, each pattern in the original collection has a domain, and this domain is restricted to include only values not included in the other patterns. It is shown that it is possible to find a minimal set of such new patterns. Case expressions over the new collection of patterns are guaranteed to behave in the same way regardless of whether the patterns are evaluated sequentially or in any other order (though this is not proven explicitly).

One practical issue with resolving sequentiality is that it might not correspond to what users might find familiar. For example, a user expecting **(case-seq)** behavior might temporarily introduce a wildcard pattern $p = \bullet$ sequentially earlier than $p' = (\texttt{nil}, \texttt{nil})$ intending to catch all instances using the first pattern. Introducing explicit markers for resolving sequentiality is one approach, but it makes programming cumbersome, and reasoning about complements of patterns (or where they might be needed) is something users would find difficult.

[1] This point due to comments by Hongwei Xi.

Particularly useful would be a tool or feature which can take a collection of patterns and produce all remaining complementary patterns. Such a tool might also highlight or point out patterns which overlap or cancel each other out (and thus might be eliminated by the pattern conversion process). This would make coding patterns much more clear for the user, and if users take advantage of this feature, it may lead to more consistent code.

Even without dependent types, the techniques used to manipulate patterns and function domains are interesting. A pattern specifies a subset of an algebraic datatype, and similar algorithms can be constructed for manipulating not only patterns, but datatype definitions. In a dependently typed setting, it would be interesting to generate a subset of a datatype which corresponds to a pattern (even if it's only to help the user determine the domain of the remaining branches of her function). Even outside a dependently typed setting, it may be useful to generate the complement of a pattern and provide the user with the subset of the datatype which corresponds to the remaining possible cases. Axioms on datatypes and concrete datatypes which are isomorphic to a datatype with an axiom defined over them can be constructed mechanically using very similar techniques.

One observation is that dependent types should make it possible to define a function with non-exhaustive patterns, and to have the type system infer a more restricted type for it (this is slightly similar to the example at the end of Section 3.2). For example, suppose our `value(t)` data type has only once constructor of type `valueint`, `ValInt`. A function which checks only one pattern, `ValInt`, could still be considered exhaustive so long as its argument type is `valueint` (this is related to using types to eliminate tag checks, as well – here, types are used to allow the user to omit certain patterns if they could not possibly occur).

The paper explores several ways in which resolving sequentiality might be useful during compilation. If the type of an expression ensures that only one constructor can be encountered, there is no need to check the tag before unwrapping it. Other ideas include using dependent types to explicitly represent the tags in a datatype. There are interesting potential generalizations of the `tag` function which is used to take a bare, untagged data value and tag it appropriately. For example, `tag` could take a datatype definition as a parameter and can then try to tag an untagged value using that datatype based solely on the structure of the value (this would only work if the structure of the domain for each constructor of the datatype is distinct). If this is done, it would make sense to define a canonical datatype for representing bare, untagged data as a collection of nodes and pointers (obviously a user can define such a datatype, but this one could be treated in a special way by the interpreter or compiler). Such an approach could potentially be tied in a consistent way to a representation of reference values on the heap, such as the one used in SML.

4 "Practical Inference for Type-Based Termination in a Polymorphic Setting" [BGP]

The authors goal is to provide an efficient type inference algorithm for a variant of system F for which the type system has been extended with annotations on types which allow for type-based checking of termination. Many definitions and concepts are drawn from earlier work in which similar extensions were added to the simply-typed lambda calculus [BFG$^+$03]. However, in this earlier work, variables inside lambda abstractions were not annotated with types. The syntax and evaluation and typing rules are provided for the language. Subject reduction is claimed for non-annotated types. No subject reduction result would be possible if annotations remained. It would be interesting to see, however, whether specifying explicitly in the evaluation rules how annotations should be substituted would resolve this issue. If each value has an explicit instantiation of its stage variable, this should certainly be the case, and would lead to a stronger result. The inference algorithm is explicitly defined, and is shown to be sound and complete with respect to the type system.

The authors point out that there is no widely used implementation of a type system which relies upon type-based termination, and that one reason for this is the limited usefulness of type-based termination for general programming languages. However, there are domains in which the termination properties of an algorithm have important implications for other properties. For example, in algorithmic mechanism design, if we wish to specify a domain-specific language for describing allocation algorithms for auctions, the divergence of an algorithm does not break a particular truthfulness property so long as whether the algorithm diverges does not depend on certain parts of the input (the bids of agents, in particular). Such a problem is slightly different from the one being addressed in the paper, but the techniques used are quite relevant, and integrating the type system from this work with one that tracks other program properties using type annotations would be useful, so that for certain annotated types, potentially divergent expressions are allowed, while for others, they are not. Alternatively, it might be possible to embed such a DSL in the language in this paper, thus completely avoiding the interaction between termination properties and other desirable algorithmic properties.

This paper also makes a contribution in a more general sense. The authors provide a type inference algorithm for a type system in which types are annotated with stage variables, but type annotations which define a subtyping relation might be used to represent many other kinds of properties (once again, I have in mind annotations which indicate whether a value is independent, monotonic, or bitonic in agent bids). It would be an interesting exercise to see which parts of the inference algorithm relate to the specific problem of checking termination, and whether the remaining parts could be parameterized by other annotation schemes.

5 "Proof Methods for Corecursive Programs" [GH00]

The authors review and provide examples for four techniques which can be used to prove algebraic properties (particularly, equalities between expressions) of corecursive functions.

Fixpoint induction involves appealing to the semantics of the underlying language. The general result shows that for a predicate which behaves appropriately with respect to the complete partial order used to represent a type, so long as two basic properties are satisfied with respect to some function f, the predicate must necessarily apply to the fix point of f. This technique is reminiscent of transfinite induction, though the similarities might be superficial. Applying the technique involves defining a property as a predicate and showing that the predicate satisfies the specified properties. Unfortunately, the implication in the technique is only in one direction, so valid predicates may exist for which this proof technique will not work.

The approximation method can reduce the problem to a proof by induction, as it is only necessary to prove that an equality holds for every finite partial list, and the proof is purely algebraic. However, the approximation function is parameterized by a natural number, and so for a more general datatype than a list there would need to be a map from naturals to that datatype which satisfies certain properties (for example, for binary trees, as the argument n grew, the function would need to traverse the tree in a breadth-first manner).

Coinduction involves the use of a bisimulation relation. The authors show that this is equivalent to the approximation method for the case of lists, and generalizing the proof of the method's validity to arbitrary datatypes would require generalizing the approximation method as well. The method is used by defining a desired candidate relation and showing it is a bisimulation. It would be interesting to try to generalize this method to more interesting classes of relations besides equality (though its validity would then need to be proven in some other manner which does not depend on the approximation method).

The fusion method is convenient because it involves defining the functions in question as unfold operations over a type, and the reasoning necessary is purely algebraic. Unfortunately, the universal property of *unfold* would look different for every datatype, though the general scheme would be the same. Once again, the validity of the technique is proven with the help of the approximation method.

It would be an interesting exercise to see whether any of the four techniques could be extended for other relations besides equality, or perhaps even for user-defined relations. For example, suppose we want to define point-wise $\leq$ on infinite lists of integers. Other interesting relations include one which says *all* the elements in an infinite list are less than *all* other elements in an infinite list, or one which says that after some *finite* number of elements are ignored, all subsequent elements in a pair of lists are related point-wise by $\leq$, or a sortedness relation which can be used to argue that merging two sorted (i.e. ascending) infinite lists results in a sorted (i.e. ascending) list.

The other obvious exercise is to extend the methods to other datatypes (or to provide a general scheme parameterized by the structure of a data type, already attempted [NGV04], though something more friendly to programmers would be preferable). Since the latter two methods both appeal to the approximation method, it seems this would involve defining an a general-purpose approximation function scheme.

6 "Fast and Loose Reasoning is Morally Correct" [DHJG06]

The goal of this paper is to show that naive algebraic proofs about program properties are still useful even in a language with partial functions. This has important implications, as algebraic reasoning about programs, especially in functional settings, is utilized frequently both when proving properties of definitions as well as when performing transformations and optimizations to the syntax within an interpreter or compiler.

The authors' strategy involves taking a limited functional language and defining two forms of semantics for that language, one total, and a domain-theoretic form allowing partial functions. The fix operator is defined only in the domain-theoretic semantics – fix is not defined in the total language, it only contains a fold and unfold operation for each inductive data type. Next, a type indexed family of partial equivalence relations (PERs) is defined on the domain-theoretic semantics, the goal being to divide the domain-theoretic semantics domains into equivalence classes. These equivalence classes can then be related to elements in the total semantics domains by providing a family of type-indexed partial surjective homomorphisms which map elements from the total semantics domain for each type to elements of the domain-theoretic semantic domains. The main result uses these homomorphisms to show that if two terms are equal under the total semantics, they are related by a PER under the domain-theoretic semantics.

The authors present an elegant variation on this result by using the PERs to define a category in which objects are types and morphisms are equivalence classes of total functions. Proofs about total terms can then take the form of commutative diagrams. This construction relies solely on the PERs and the domain-theoretic semantics, and does not refer to the total semantics. If this technique is used to reason about programs, it should be possible to reason about the fix operator, something which cannot be done by appealing to the total semantics.

The results in this work are limited in a number of ways. Total reasoning with general recursive definitions is not possible, and is limited to the fold and unfold operators on datatypes. However, fold and unfold operations are particularly amenable to equational reasoning, and are sometimes a good way to reason even in a partial setting (as demonstrated by the authors' last example in Section 10). Furthermore, the language under consideration is not polymorphic, though one of the initial examples presented for motivation involves the map and reverse functions for lists, which are typically polymorphic. It would be interesting to find out whether the argument about reasoning in a partial language can be extended to a polymorphic setting.

Another context in which the results might be relevant are when creating interactive proof assistants. In such a setting, the necessary premises are all made explicit by the assistant and must be handled by the user. This work suggests that some of the premises which relate to the partiality of functions could be discharged automatically. It would be interesting to see whether this could make a proof assistant designed for equational reasoning less cumbersome to use. In fact, this paper provides a way to relate a partial language to a total language for which we may already have proof assistants, allowing programmers to write proofs about partial programs which are still arguably applicable.

References

[BFG+03] G. Barthe, M. Frade, E. Gimenez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions, 2003.

[BGP] Gilles Barthe, Benjamin Gregoire, and Fernando Pastawski. Practical inference for typed-based termination in a polymorphic setting.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.

[DHJG06] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–217, New York, NY, USA, 2006. ACM Press.

[GH00] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. 2000.

[NGV04] Tarmo Uustalu Neil Ghani and Varmo Vene. Build, augment, destroy. universally. In *Proceedings of Programming Languages and Systems: Second Asian Symposium, APLAS, 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 327–341. Springer Verlag, 2004.

[Rey] John C. Reynolds. What do types mean? - from intrinsic to extrinsic semantics.

[Xi99] H. Xi. Some practical aspects of dependent datatypes, 1999. Available as http://www.ececs.uc.edu/ hwxi/ academic/papers/PADD.ps.