Data Types with Negation (Talk Abstract)

Robert Atkey
University of Strathclyde*, Glasgow, UK
robert.atkey@strath.ac.uk

1 Introduction

Inductive data types are a foundational tool for representing data and knowledge in dependently typed programming languages. The user provides a collection of rules that determine positive evidence for membership in the type. Elimination of an inductive type corresponds to structural induction on its members. For example, in a language like Agda we can define inductive data types for representing evidence that natural numbers are odd or even:

$$\begin{array}{l}\textbf{data}\,\mathrm{Even} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{zero\text{-}even} : \mathrm{Even}\,\mathsf{zero}\\ \quad \mathsf{succ\text{-}odd} : \forall x \to \mathrm{Odd}\,x \to \mathrm{Even}\,(\mathsf{succ}\,x)\end{array} \qquad \begin{array}{l}\textbf{data}\,\mathrm{Odd} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{succ\text{-}even} : \forall x \to \mathrm{Even}\,x \to \mathrm{Odd}\,(\mathsf{succ}\,x)\end{array}$$

But what if our data modelling requires negative information as well as positive? One example is this alternative encoding of evenness, which marks a number as even if its predecessor is not:

$$\begin{array}{l}\textbf{data}\,\mathrm{Even} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{zero\text{-}even} : \mathrm{Even}\,\mathsf{zero}\\ \quad \mathsf{succ\text{-}not} : \forall x \to \neg(\mathrm{Even}\,x) \to \mathrm{Even}\,(\mathsf{succ}\,x)\end{array}$$

Here we have used a negation operator $\neg(-)$ to represent the negation of $\mathrm{Even}\,x$ instead of an explicit definition of oddness. A example that makes more essential use of negation is a data type representing successful parses from a backtracking Parsing Expression Grammar (PEG) parser. PEGs use an ordered choice operator `A / B` which means "try to parse as `A`, and if that fails then try to parse a `B`". If we represent evidence of successful parses as elements of a data type indexed by the string to be parsed and the suffix remaining to be parsed, then we could use a data type with negation to represent a PEG production of the form `P ::= A / B`:

$$\begin{array}{l}\textbf{data}\,\mathrm{ParseP} : \mathrm{String} \to \mathrm{String} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{choice1} : \forall i\,o.\,\mathrm{ParseA}\,i\,o \to \mathrm{ParseP}\,i\,o\\ \quad \mathsf{choice2} : \forall i\,o.\,\neg(\exists o'.\,\mathrm{ParseA}\,i\,o') \to \mathrm{ParseA}\,i\,o \to \mathrm{ParseP}\,i\,o\end{array}$$

So a $\mathrm{ParseP}\,i\,o$ is constructible if a $\mathrm{ParseA}\,i\,o$ is possible or if that is *not* possible for any left over o' and a $\mathrm{ParseB}\,i\,o$ is possible.

If we interpret negation as $\neg A = A \to \bot$ then a proof assistant like Agda will reject these definitions. And for good reason: a data type using this kind of negation does not define a monotone operator on sets of evidence, and so does not have a well defined least fixpoint. However, it is possible to give these types a reasonable semantics. We will take two steps. First, we need a constructive notion of negation, so that we have explicit evidence of refutations to work with. Second, we need to determine the semantics of recursive negation, which we will do using the 3-valued stable model semantics of logic programming.

*Research funded by EPSRC Grant EP/T026960/1 *AISEC: AI Secure and Explainable by Construction*.

Submitted to:
MSFP 2022

© R. Atkey
This work is licensed under the
Creative Commons Attribution License.

2 Constructive Negation

To give a constructive semantics of negation, we will construct evidence for refutations of propositions simultaneously with evidence for their proofs. Negation of a proposition is then nothing more than the swapping of the status of proofs and refutations. Concretely, we define $\mathrm{Set}^{\pm} = \mathrm{Set} \times \mathrm{Set}$ with the following constructions of propositions, directly witnessing the de Morgan dualities from classical logic:

$$(A^+,A^-) \times (B^+,B^-) = (A^+ \times B^+, A^- + B^-) \qquad \Sigma x : X.\,(A^+[x],A^-[x]) = (\Sigma x : X\, A^+[x], \Pi x : X.\, A^-[x])$$
$$(A^+,A^-) + (B^+,B^-) = (A^+ + B^+, A^- \times B^-) \qquad \neg(A^+,A^-) = (A^-,A^+)$$

Entailment in $\mathrm{Set}^{\pm}$ goes forwards in the first component and backwards in the second (so we are working in the category $\mathrm{Set} \times \mathrm{Set}^{op}$, a simple form of Chu Spaces [1], a model of Linear Logic).

We can construct models of inductive data types in $\mathrm{Set}^{\pm}$ for positive functors F, where the proof and refutation parts are independent, by taking the initial algebra in the first component and the final coalgebra in the second. However, this does not immediately help us with data types that involve negation.

3 Stable Semantics of Negation

To model datatypes with negation, we turn to ideas from Logic Programming, where the semantics of logic programs that involve negation has been the subject of intense study. One popular semantics is well-founded or 3-valued stable semantics. Intuitively, this semantics interprets negation in terms of lack of support for a proposition given the rules defining the program. This semantics has a fixed point characterisation [2] that we can replicate in terms of data types as follows.

If we have a data type with negation (assuming no indexing for now), we can derive an operator $F(A^+,A^-) = (F^+(A^+,A^-), F^-(A^+,A^-))$ using the interpretations of the connectives given above. This does not necessarily have a fixed point in $\mathrm{Set}^{\pm}$. However, if we assume that we have some fixed (X^+,X^-) we can relatise F to X to give a functor $(F/X)(A^+,A^-) = (F^+(A^+,X^-), F^-(X^+,A^-))$, which makes the proofs and refutations independent, and so does have a least fixpoint $\mu F/X$ in $\mathrm{Set}^{\pm}$. This construction functorial in X *covariantly* in both components, so there is a least fixpoint in $\mathrm{Set} \times \mathrm{Set}$, yielding the overall semantics:

$$\mu(X^+,X^-).(\mu A^+.F^+(A^+,X^-), \nu A^-.F^-(X^+,A^-))$$

Intuitively, we build up the semantics of a data type in stages. We start with no negative information about proofs or refutations, and derive all the positive proofs and refutations we can from this. Swapping these proofs and refutations, we use them as the negative information in the next round, and so on.

It is now possible to compute models for all data types with negation. One example is the "liar" type:

$$\textbf{data}\ \mathrm{Liar} : \mathrm{Set}\ \textbf{where}\ \mathsf{liar} : \neg\mathrm{Liar} \to \mathrm{Liar}$$

This type is assigned the model $(\bot,\bot)$ – there are no proofs and no refutations of this type.

We have now built a plausible semantics for data types with negation. It remains to be seen how best to accomplish reasoning over inhabitants of data types with negative information.

References

[1] Po-Hsiang Chu (1978): *Constructing $*$-autonomous categories*. Master's thesis, McGill University.

[2] Teodor C. Przymusinski (1990): *The Well-Founded Semantics Coincides with the Three-Valued Stable Semantics*. *Fundam. Inform.* 13(4), pp. 445–463.