COMPUTATIONAL ABSTRACTION

RAYMOND TURNER

The practice of Computer Science is dominated by various processes or devices of abstraction. Many these devices are built into specification and programming languages. Indeed, they are the mechanisms of language design, and the process of abstraction maybe seen as generating new languages from given ones. Our objective in this paper is to provide a logical analysis of such abstraction.

Abstraction in mathematics has been a topic of philosophical concern since antiquity. Moreover, much contemporary research inspired by Frege (1884) has heralded a new and promising episode. However, this work, that generally goes by the name *the way of abstraction*, has mostly been aimed at classical logic and mathematics where the ultimate goal has been to *abstract* the axioms of Zermelo-Fraenkel set theory. Little work has been aimed at other foundational frameworks such as type theory, the central carrier of computational abstraction. Our intention is to explore how the way of abstraction may provide a foundational framework for the latter.

Frege observes that many of the singular terms that appear to refer to abstract entities are formed by means of functional expressions. While many singular terms formed by means of such expressions denote ordinary concrete objects, the functional terms that pick out abstract entities are distinctive in the sense that they are picked out by functional expression of the following forms.

(1) *The direction of a line* A *=The direction of line* B *iff* $A||B$.
(2) *The number of books on the shelf = The number of plates on the table iff* $BS \approx PT$

where $A||B$ asserts that A is parallel to B and $BS \approx PT$ that that there is a one-to-one correspondence between the collection of books on the shelf and the collection of plates on the table. Inspired by such examples, an abstraction principle may be formulated as a universally quantified bi-conditional of the following form:

$$\forall x.\forall y.(f(x) = f(y)) \leftrightarrow R(x, y)$$

where x and y are variables of some logical language, f is a term forming operator and R is an equivalence relation.

In general, the abstractionist views such abstraction principles as introducing a new *kind* of object. For example,

The direction of a line A *=The direction of line* B *iff* $A||B$

introduces the kind of things that are *directions*. But notice that these abstractions also require some a *kind of thing* on which the abstraction carves out the ones that are equivalent. In other words, implicit in the discussion is a further kind of object - the domain that the equivalence relation operates on. For example, in the case of directions the domain of abstraction is the kind of *lines* and in the case of numbers some notion of *collection* is presupposed. Consequently, we have the following ontological demand that insists that all new abstracts emerge from some source

kind.

$$\forall x : T_{/R} \centerdot \exists y : T \centerdot x = f(y)$$

Here $T_{/R}$ is the new abstracted domain and T the source domain of abstraction. On this view, abstraction results in a new kind of object that emerges from an old one. Such an approach to the development of type theories views abstraction as a way of creating new languages and type theories from existing ones.

Based upon these observations we provide an approach to abstraction aimed at the development of type theories. It avoids the *bad company* paradox in a way that parallels Russell's way out by appeal to some notion of type. However, the way of abstraction does not just postulate a given class of types and type constructors as Russell did. Abstraction offers a dynamic for the introduction of new types and type constructors. Indeed, we are not restricted to the types of Russell's simple type theory - or any other given type theory. We can explore the rich and varied notions of type that are to be found in modern computer science. The only restriction is that these new types emerge via the way of abstraction. On this approach every proposition of the new language that gives rise to an equivalence relation on some given type, gives rise to a new type of object. So that the way of abstraction now takes the following shape.

$$\forall x : T. \forall y : T. (f(x) =_{T_{/R}} f(y)) \leftrightarrow R(x, y)$$

where T is an existing type, R is a relation determined by some well-formed formulae of the formal language, $T_{/R}$ is the new type formed by elements of the form $f(t)$ where $t : T$, and $=_{T_{/R}}$ the equality for the new type. This is exactly the set-up required to describe computational abstraction.

Observe that, unlike Russells' type theory, types are constructed in a predicative way. More explicitly, new types arise by abstracting on given types, and there is no quantification over the new type involved in the specification of the relation R: the quantification over the new type is not available until after the new type has been introduced. The distinctive feature of successful abstraction is that every question about the abstracts is interpreted as a question about the entities on which we abstracted. This yields a conservative extension result for the new theory. The conservative extension result provides an implementation of the new theory in the old one. Consequently, the intuition that implementation and abstraction are orthogonal activities is formally explained.

Finally, the different *intentional stances* adopted in implementation and abstraction throw light on the wrong, yet seemingly persuasive, perspective that language implementations provide semantic interpretations.