Science of Computer Programming 13 (1989/90) 73-110
North-Holland

ELEMENTARY DATA STRUCTURES IN ALGOL-LIKE LANGUAGES*

R.D. TENNENT
Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada

Communicated by D. Bjørner
Received August 1988
Revised May 1989

Abstract. J.C. Reynolds has pointed out that ALGOL 60 has a set of properties not shared by most of the languages usually regarded as being its successors. We propose to use this ALGOL-like framework to design a language that could adequately support *both* applicative *and* imperative programming while *also* retaining the advantages of each of the "pure" frameworks. This paper discusses elementary data-structuring facilities (products, arrays, sums) for such a language, taking advantage of recent developments, such as this author's "quantification" notation, and the notion of "conjunctive type" proposed by Coppo and Dezani, and adapted to explicitly-typed languages by Reynolds.

1. Introduction

ALGOL 60 is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors.

C.A.R. Hoare

The *applicative* (or functional) programming paradigm [4, 8, 25] evolved from (pure) LISP and emphasized

(i) the use of recursive data types (such as lists and trees) and recursively-defined and higher-order functions;
(ii) the succinctness and expressiveness possible without explicit types; and
(iii) the ease of reasoning about expressions without side-effects and about lambda calculus-based procedures.

The *imperative* (or procedural) programming paradigm [12, 19, 20, 40, 66] evolved from FORTRAN, and emphasized

(i) the use of selectively-updateable data structures (such as arrays and records) and iteration;

* This work was supported in part by an operating grant from the Natural Sciences and Engineering Research Council of Canada, and by a grant from the Information Technology Research Centre of Ontario.

(ii) the efficiency and security obtainable with static types and static or stack-oriented storage management; and

(iii) the ease of reasoning about simple imperative algorithms when there is no aliasing.

Many language designers have attempted to find a compromise between these extreme positions. Even some very early design efforts [1, 5, 63] did, in retrospect, attempt to integrate applicative and imperative programming, and [6, 13, 21–23, 26, 32, 39, 64] are more recent discussions and proposals.

Is it possible for a language to support *both* programming styles adequately, and also *combine* the advantages of the "pure" frameworks? Ideally, such a programming language would

- provide recursion, higher-order procedures, and recursively-defined types, and *also* assignment, iteration and selectively-updateable data structures;
- have the flexibility and syntactic succinctness of typeless languages, and *also* provide the security and efficiency of statically-typed languages; and
- allow simple reasoning about properties of both imperative and applicative aspects of programs and *also* have simple and efficient implementations.

In the author's opinion, no existing or proposed programming language achieves this ambitious ideal. The procedure mechanisms in most imperative languages are much more complicated and restrictive than those in well-designed purely-applicative languages. The ease of reasoning possible with a purely-applicative language is typically lost when imperative features are added. Similarly, the ease of reasoning possible using Hoare-like axioms with a simple imperative language is typically lost when procedures are added. Nevertheless, there have been several important developments in recent years.

The *call-by-need* form of parameter passing (also known as "lazy evaluation") [14, 15, 65] makes it possible to obtain the attractive substitution-oriented semantics of call-by-name with (almost) the efficiency of call-by-value, and there has been considerable work [2] on "strictness analysis" with the same motivation. The language ML [29] demonstrates that it is possible to use type inference and polymorphism to achieve the efficiency and security of statically-typed languages with (almost) the expressiveness and succinctness of typeless languages.

Unfortunately, none of these ideas has yet been used very successfully in languages that adequately support imperative programming. For example, in [61], the most recent attempt to extend the polymorphic type discipline of [11] to a language with imperative features, it is noted that this problem has proved to be "an obstacle (if not *the* obstacle) to the harmonic integration of imperative and functional language features," and there are further difficulties with incorporating implicit conversions such as the "de-referencing" coercion from variables to expressions.

In the author's opinion, the most important contribution relevant to providing support for both applicative and imperative programming in a single language is the observation by Reynolds [46] that ALGOL 60 has a distinctive set of properties

not shared by most of the languages usually regarded as being its successors, including ALGOL 68 [63] and PASCAL [66]. The principles that Reynolds suggests are characteristic of "ALGOL-like languages" are described there as follows.

(1) The order of evaluation for parts of expressions, and of implicit conversions between data or phrase types, is indeterminate, but the meaning of the language, at an appropriate level of abstraction, is independent of this indeterminacy.

In short, expressions must be "mathematical," in that evaluation does not result in side-effects or jumps, even if the language has imperative features such as assignments and escapes. The following statement, from Naur [33], can be construed as justification for the claim that ALGOL 60 was intended to have this property: "a numerical (Boolean) expression is a rule for computing a numerical (logical) value," with nothing said about side-effects or jumps; see also [24].

(2) Algol is obtained from the simple imperative language [of **while** programs] by imposing a procedure mechanism based on a fully-typed, call-by-name lambda calculus.

Although many languages are *notationally* close to the lambda calculus, only ALGOL 60 (with its "copy rule") and its very close relations are faithful to the equational *laws* of the lambda calculus. In most languages, the provision of higher-order procedures has drastic implementation consequences; but not in ALGOL-like languages.

(3) The language obeys a stack discipline.

A semantics for variable declarations which captures this discipline in an abstract way is described in [34, 35, 46, 58]. The reason that ALGOL 60 could provide higher-order procedures without requiring complex and artificial restrictions to maintain stack-implementability for local variables is that its type structure makes an important distinction.

(4) There are two fundamentally different kinds of type: *data types*, each of which denotes a set of values appropriate for certain variables and expressions, and *phrase types*, each of which denotes a set of meanings appropriate for certain identifiers and phrases.

In ALGOL 60, as first pointed out by Strachey [55], elements of data types (such as truth values and numbers) are assignable (to variables), but not denotable (by identifiers), whereas elements of phrase types (such as variables and procedures) are denotable but not assignable; the latter property allows stack-implementability. Finally, the presence of both conditional statements and conditional expressions in ALGOL 60 suggests the following principle, which is a generalization of the principles of abstraction and qualification of Tennent [56].

(5) Facilities such as procedure definition, recursion, and conditional and case constructions are uniformly applicable to all phrase types.

Furthermore, Reynolds has shown that it is possible to reason straightforwardly about both applicative and imperative aspects of programs in ALGOL-like languages using "specification logic" [45, 47], a formal system that *combines*

- Hoare-like axioms for reasoning about commands,
- first-order theories of the data types for reasoning about expressions,
- fixed-point induction for reasoning about recursion, and
- rules derived from the laws of the lambda calculus for reasoning about procedures.

In particular, the imperative mechanisms of an ALGOL-like language do not interfere at all with applicative properties, which is certainly not the case with languages such as ML, in which imperative features have been added to a lambda calculus-based applicative notation in a completely different way. It is true that procedures do affect Hoare-like reasoning about imperative features, but Reynolds shows that this is manageable by introducing a suitable concept of "non-interference" into the specification language. See [58] for discussion of some problematical aspects of the interpretation of specification logic.

Reynolds has also described an elegant approach to *syntactic* control of interference for ALGOL-like languages [43]. If this can be perfected, it should simplify Hoare-like reasoning about most programs and will also allow a compiler to recognize easily that call-by-need is a correct implementation of the semantics of call-by-name, even in the presence of imperative features.

In summary, it appears that, far from being the completely obsolete language that most programmers and language designers have long thought it to be, ALGOL 60 uniquely *combines* desirable attributes of both purely-applicative and conventional imperative languages, and provides us with a framework for designing a language that will approach the ambitious ideal stated above.

In this paper we take some steps towards the goal of designing modern ALGOL-like languages in the sense of Reynolds by describing the (abstract) syntax (including type and scope constraints) and (denotational) semantics of the elementary data-structuring facilities that seem to be appropriate to such languages. The main reason that ALGOL 60 was superseded by PASCAL and derivatives after the appearance of [18] is its inadequate data-structuring facilities. Some suggestions about *phrase-type* structuring are given by Reynolds [46], but *data-type* structures are also needed. We will also see that the interpretation of the phrase-type sums described in [46] is rather problematical. Finally, we can now take advantage of two developments that have been made since [46]: the "quantification" notation described by Tennent [57], and the adaptation to explicitly-typed languages by Reynolds [50] of the notion of "conjunctive type" originally proposed by Coppo and Dezani [10].

This paper is organized as follows. Sections 2–5 summarize the necessary background on types, syntax, semantics, coercions, and conjunctive types; none of this material is new. Sections 6 and 7 discuss phrase-type and data-type structures, respectively. Section 8 outlines the areas where further research is needed. For reasons of presentation, the language is described incrementally: a basic language is presented first, and then augmented in succeeding sections.

2. Types

Our initial type structure is determined by the following productions:

data types:

$\tau ::= \mathbf{bool}$	Booleans,
$\mid \mathbf{nat}$	natural numbers;

phrase types:

$\theta ::= \mathbf{exp}[\tau]$	expressions,
$\mid \mathbf{comm}$	commands,
$\mid \mathbf{acc}[\tau]$	acceptors,
$\mid \mathbf{var}[\tau]$	variables,
$\mid \theta \to \theta'$	procedures.

Acceptors are the "updating" or "write-only" components of variables, sometimes termed "*l*-values." The result type θ' of a procedural type $\theta \to \theta'$ may be *any* of the phrase types. If the result type is **comm**, the procedures are conventional "subroutines" or "proper" procedures. If the result type of a procedural type is $\mathbf{exp}[\tau]$, the procedures are "functions" whose calls are expressions for values of type τ. There are also variable-returning and acceptor-returning procedures (sometimes called "selectors") and procedure-returning procedures. Similarly, the argument type of a procedural type may be any of the phrase types. For example, a parameter of type $\mathbf{acc}[\tau]$ is a kind of pure result parameter. Multi-parameter procedures will be added as syntactic sugar in the following section.

To simplify the discussions of semantics in this paper, we consider only a *fixed* set S of storage states. The generalization appropriate to treat features for which this is not sufficient (variable declarations and block expressions) is discussed in [58–60]. Also, we avoid features (such as escapes) that require the use of continuations, so that we can give a "direct" semantics. Finally, we assume that commands are deterministic.

Each data type τ denotes a set $[\![\tau]\!]$ of values possible for some kind of expression or variable:

$[\![\mathbf{bool}]\!] = \{true, false\}$	truth values,
$[\![\mathbf{nat}]\!] = \{0, 1, 2, \ldots\}$	natural numbers.

Each phrase type θ denotes a directed-complete partially-ordered set $[\![\theta]\!]$ of meanings

possible for phrases of that type. We term these ordered structures "domains," but do not insist on the existence of a least element or algebraicity. For domains D and D', let

- $D_\perp$ be D "lifted" by the addition of a new least element $\perp$,
- $D \times D'$ be the Cartesian product of D and D', ordered component-wise,
- $D \to D'$ be the set of continuous functions from D to D' ordered point-wise,
- $D \rightsquigarrow D'$ be the domain of continuous[1] *partial* functions from D to D' ordered pointwise; i.e., $f \sqsubseteq_{D \rightsquigarrow D'} g$ iff, for all $x \in D$, if $f(x)$ is defined, then $g(x)$ is defined and $f(x) \sqsubseteq_{D'} g(x)$.

In the generalization discussed in [58-60], $D \rightsquigarrow D'$ is not, in general, isomorphic to $D \to D'_\perp$. The domains denoted by the phrase types above are defined inductively as follows:

$$[\![\mathbf{exp}[\tau]]\!] = S \to [\![\tau]\!]_\perp,$$

$$[\![\mathbf{comm}]\!] = S \rightsquigarrow S,$$

$$[\![\mathbf{acc}[\tau]]\!] = [\![\mathbf{exp}[\tau]]\!] \to [\![\mathbf{comm}]\!],$$

$$[\![\mathbf{var}[\tau]]\!] = [\![\mathbf{acc}[\tau]]\!] \times [\![\mathbf{exp}[\tau]]\!],$$

$$[\![\theta \to \theta']\!] = [\![\theta]\!] \to [\![\theta']\!],$$

where sets such as S and $[\![\tau]\!]$ should be regarded as discretely-ordered domains (i.e., $x \sqsubseteq y$ iff $x = y$). Thus, a phrase of type $\mathbf{exp}[\tau]$ yields a meaning that, when provided with a storage state, produces an element of $[\![\tau]\!]$ as its value (or aborts or fails to terminate, both modelled by the least element $\perp$). A command meaning is a partial function on states. The use of total functions to a "lifted" domain with expressions but partial functions to a set with commands is essential in the more general semantic framework discussed in [58-60]. A variable consists of an acceptor component (*l*-value) and an expression component (*r*-value); that is, variables are like the "load-update pairs" of Strachey [54] and Park [37] and the "implicit references" of GEDANKEN (Reynolds [42]). From now on, $\mathbf{acc}[\tau]$ will be regarded as being an *abbreviation* for the procedural type $\mathbf{exp}[\tau] \to \mathbf{comm}$. Notice that, for every phrase type θ, $[\![\theta]\!]$ has a least element; this will be denoted by $\perp_\theta$.

3. Phrases

We will specify the (abstract) syntax of phrases in our example language by using "compositional" (and possibly-schematic) inference rules for formulas of the form $Z : \theta$ (i.e., Z is a well-formed phrase of type θ).

[1] We will only use this construction with domains D for which *all* partial functions are continuous, but a general definition may be found in [38].

Here is a selection of rules for expressions:

zero:

$$\frac{}{\mathbf{0}:\mathbf{exp[nat]}};$$

successor:

$$\frac{N:\mathbf{exp[nat]}}{\mathbf{succ}\ N:\mathbf{exp[nat]}};$$

ordering:

$$\frac{N_0:\mathbf{exp[nat]} \quad N_1:\mathbf{exp[nat]}}{N_0 < N_1:\mathbf{exp[bool]}};$$

conjunction:

$$\frac{B_0:\mathbf{exp[bool]} \quad B_1:\mathbf{exp[bool]}}{B_0\ \mathbf{and}\ B_1:\mathbf{exp[bool]}};$$

equality:

$$\frac{E_0:\mathbf{exp}[\tau] \quad E_1:\mathbf{exp}[\tau]}{E_0 = E_1:\mathbf{exp[bool]}}.$$

Notice that the equality rule is *schematic* over all data types τ; however, both operands must be expressions for the *same* data type.

Here are some rules for commands:

null:

$$\frac{}{\mathbf{skip}:\mathbf{comm}};$$

assignment:

$$\frac{A:\mathbf{acc}[\tau] \quad E:\mathbf{exp}[\tau]}{A := E:\mathbf{comm}};$$

sequencing:

$$\frac{C_0:\mathbf{comm} \quad C_1:\mathbf{comm}}{C_0; C_1:\mathbf{comm}};$$

iteration:

$$\frac{B:\mathbf{exp[bool]} \quad C:\mathbf{comm}}{\mathbf{while}\ B\ \mathbf{do}\ C:\mathbf{comm}}.$$

Finally, we give rules that are schematic over *phrase* types:

bracketing:

$$\frac{Z:\theta}{(Z):\theta};$$

conditional:

$$\frac{B:\mathbf{exp[bool]} \quad Z_1:\theta \quad Z_2:\theta}{\mathbf{if}\ B\ \mathbf{then}\ Z_2\ \mathbf{else}\ Z_2:\theta};$$

application:

$$\frac{P:\theta\to\theta' \quad Q:\theta}{PQ:\theta'};$$

abstraction:

$$\frac{\begin{array}{c}[\iota:\theta]\\ \vdots\\ P:\theta'\end{array}}{\lambda\iota:\theta.P:\theta\to\theta'}.$$

The rule for abstraction is in natural-deduction format [41, 62]; that is, the formal system allows us to assert "deducibility statements" of the form $\pi\vdash Z:\theta$, where π is a *phrase-type assignment*, a finite set of assumptions of the form $\iota:\theta$, for distinct ι. The rule for application should therefore be interpreted as stating that, for any type assignment π, if $\pi\vdash P:\theta\to\theta'$ and $\pi\vdash Q:\theta$, then $\pi\vdash PQ:\theta'$, and similarly for all of the preceding rules.

We can regard a type assignment π as being a function, so that, if its domain is $dom(\pi)$, then, for all $\iota\in dom(\pi)$, $\pi(\iota)$ is the assumed type of ι. The rule for abstraction then states that, for any π, if $(\pi\,|\,\iota\mapsto\theta)\vdash P:\theta'$, then $\pi\vdash\lambda\iota:\theta.P:\theta\to\theta'$, where $(\pi\,|\,\iota\mapsto\theta)$ denotes the type assignment π' such that $dom(\pi')=dom(\pi)\cup\{\iota\}$, $\pi'(\iota)=\theta$, and $\pi'(\iota')=\pi(\iota')$ for all $\iota'\in dom(\pi)$ such that $\iota'\neq\iota$. The deducibility of $\iota:\theta$ from any π such that $\pi(\iota)=\theta$ is implicit in the natural-deduction framework.

We specify semantics by means of "semantic equations" which give a compositional interpretation of each construct solely in terms of the interpretations of its immediate components. For any type assignment π, let $[\![\pi]\!]$ be $\prod_{\iota\in dom(\pi)}[\![\pi(\iota)]\!]$; i.e., the set of all "environment" functions u with the same domain as π and satisfying the constraint that, for all $\iota\in dom(\pi)$, $u(\iota)\in[\![\pi(\iota)]\!]$. Then we define valuations $[\![\cdot]\!]_{\pi\theta}$ mapping any phrase Z such that $\pi\vdash Z:\theta$ into an element of $[\![\pi]\!]\to[\![\theta]\!]$. The π and θ subscripts on phrase-valuation brackets will usually be omitted except when there is a possibility of confusion. Here are some typical semantic equations for expressions and commands, using u and s to range over $[\![\pi]\!]$ and S, respectively:

$$[\![\mathbf{0}]\!]us = 0,$$

$$[\![\mathbf{succ}\ N]\!]us = \begin{cases} [\![N]\!]us + 1, & \text{if } [\![N]\!]us \neq \perp, \\ \perp, & \text{if } [\![N]\!]us = \perp, \end{cases}$$

$$[\![\mathbf{skip}]\!]u = id_S,$$

$$[\![A := E]\!]u = [\![A]\!]u([\![E]\!]u),$$

$$[\![C_0; C_1]\!]u = [\![C_0]\!]u ; [\![C_1]\!]u,$$

where id_S is the identity function on S and ";" on the right-hand side denotes composition of partial functions in diagrammatic order; i.e., $(f; g)(s)$ is $g(f(s))$ if $f(s)$ is defined, and is undefined otherwise.

The equations for identifiers and procedural application and abstraction are as follows:

$$[\![\iota]\!]u = u(\iota),$$

$$[\![PQ]\!]u = [\![P]\!]u([\![Q]\!]u),$$

$$[\![\lambda\iota : \theta . P]\!]ua = [\![P]\!](u \mid \iota \mapsto a),$$

where $a \in [\![\theta]\!]$ and the interpretation of $(u \mid \iota \mapsto a)$ is analogous to that of $(\pi \mid \iota \mapsto \theta)$. This is a "call-by-name" semantics, in that if an actual parameter is an expression, variable or command, it is not evaluated or executed in the application. The usual substitution law for the lambda calculus [53] holds (in typed form) for *all* phrases: for any appropriate environment u,

$$[\![P[Q/\iota]]\!]u = [\![P]\!](u \mid \iota \mapsto [\![Q]\!]u),$$

where $P[Q/\iota]$ denotes the result of substituting Q (of appropriate type) for all free occurrences of ι in P (with the usual changes of bound identifiers in P to avoid capturing free identifiers of Q).

The conditional is interpreted as follows:

$$[\![\mathbf{if}\ B\ \mathbf{then}\ Z_1\ \mathbf{else}\ Z_2]\!]_{\pi\theta}(u) = cond_\theta([\![B]\!]u, [\![Z_1]\!]_{\pi\theta}(u), [\![Z_1]\!]_{\pi\theta}(u)),$$

where the auxiliary functions $cond_\theta : [\![\mathbf{exp[bool]}]\!] \times [\![\theta]\!] \times [\![\theta]\!] \to [\![\theta]\!]$ are defined by induction on θ as follows:

$$cond_{\mathbf{comm}}(b, c_1, c_2)(s) = \begin{cases} c_1(s), & \text{if } b(s) = \mathit{true}, \\ c_2(s), & \text{if } b(s) = \mathit{false}, \\ \text{undefined}, & \text{if } b(s) = \perp, \end{cases}$$

$$cond_{\mathbf{exp}[\tau]}(b, e_1, e_2)(s) = \begin{cases} e_1(s), & \text{if } b(s) = \mathit{true}, \\ e_2(s), & \text{if } b(s) = \mathit{false}, \\ \perp, & \text{if } b(s) = \perp, \end{cases}$$

$$cond_{\theta \to \theta'}(b, p_1, p_2)(a) = cond_{\theta'}(b, p_1(a), p_2(a)) \quad \text{for all } a \in [\![\theta]\!],$$

$$cond_{\mathbf{var}[\tau]}(b, \langle a_1, e_1\rangle, \langle a_2, e_2\rangle) = \langle cond_{\mathbf{acc}[\tau]}(b, a_1, a_2), cond_{\mathbf{exp}[\tau]}(b, e_1, e_2)\rangle,$$

where $cond_{\mathbf{acc}[\tau]} = cond_{\mathbf{exp}[\tau] \to \mathbf{comm}}$.

Many constructions can be treated as "syntactic sugar" [25], that is to say, as abbreviations of combinations of more-basic constructs. For example, if, as suggested by Schönfinkel [51], a type of the form $\theta_1 \times \cdots \times \theta_n \to \theta$ is treated as an alternative notation for $\theta_1 \to \cdots \to \theta_n \to \theta$ (where the arrow associates to the right), then we can provide multi-parameter procedures as follows:

multiple abstraction:

$$\frac{\begin{array}{c}[\iota_i : \theta_i, \text{ for } i = 1, 2, \ldots, n] \\ \vdots \\ P : \theta\end{array}}{\lambda(\iota_1 : \theta_1, \ldots, \iota_n : \theta_n).P : \theta_1 \times \cdots \times \theta_n \to \theta}$$

multiple application:

$$\frac{P : \theta_1 \times \cdots \times \theta_n \to \theta \quad Q_i : \theta_i, \text{ for } i = 1, 2, \ldots, n}{P(Q_1, \ldots, Q_n) : \theta},$$

by adopting the following "de-sugaring" equivalences:

$$\lambda(\iota_1 : \theta_1, \ldots, \iota_n : \theta_n).P \equiv \lambda\iota_1 : \theta_1 \ldots . \lambda\iota_n : \theta_n . P,$$

$$P(Q_1, \ldots, Q_n) \equiv P(Q_1) \cdots (Q_n).$$

Convenient notation for making local definitions was suggested by Landin [25]:

definition:

$$\frac{\begin{array}{c}[\iota : \theta] \\ \vdots \\ P : \theta \quad Q : \theta'\end{array}}{\textbf{let } \iota \textbf{ be } P \textbf{ in } Q : \theta'}$$

with the de-sugaring

$$\textbf{let } \iota \textbf{ be } P \textbf{ in } Q \equiv (\lambda\iota : \theta . Q)P.$$

Note that the type of the bound identifier need not be specified explicitly in the definition because it may be *inferred* from the type of P. It can easily be verified that this de-sugaring implies the following valuation:

$$[\![\textbf{let } \iota \textbf{ be } P \textbf{ in } Q]\!]u = [\![Q]\!](u \,|\, \iota \mapsto [\![P]\!]u)$$

and so, by the substitution law,

$$\textbf{let } \iota \textbf{ be } P \textbf{ in } Q \equiv Q[P/\iota].$$

For defining *procedures*, it seems preferable to introduce another notation suggested by Landin because it puts the "low-level" code that defines the procedure after the

"high-level" code that uses it:

procedure definition:

$$\frac{\begin{array}{cc}[\iota_i:\theta_i,\ \text{for } i=1,2,\ldots,n] & [\iota:\theta_1\times\cdots\times\theta_n\to\theta] \\ \vdots & \vdots \\ P:\theta & Q:\theta'\end{array}}{\textbf{let } \iota:\theta_1\times\cdots\times\theta_n\to\theta \textbf{ in } Q \textbf{ where } \iota(\iota_1,\ldots,\iota_n)=P:\theta'}$$

with the de-sugaring

$$\textbf{let } \iota:\theta_1\times\cdots\times\theta_n\to\theta \textbf{ in } Q \textbf{ where } \iota(\iota_1,\ldots,\iota_n)=P$$
$$\equiv(\lambda\iota:\theta_1\times\cdots\times\theta_n\to\theta.Q)(\lambda(\iota_1:\theta_1,\ldots,\iota_n:\theta_n).P).$$

There are straightforward generalizations of these definition forms to allow multiple ("simultaneous") definitions.

Another class of syntactic sugarings [57] is based on the notation for quantification in predicate logic:

quantification:

$$\frac{\begin{array}{c}[\iota:\theta'] \\ \vdots \\ Q:(\theta'\to\theta'')\to\theta \quad P:\theta''\end{array}}{\#Q\iota.P:\theta}$$

with the de-sugaring

$$\#Q\iota.P\equiv Q(\lambda\iota:\theta.P).$$

Again, the type of the bound variable can be inferred from the type of Q. For example, if C is a command and, for any data type τ,

$$\textbf{new}[\tau]:(\textbf{var}[\tau]\to\textbf{comm})\to\textbf{comm}, \qquad \textbf{valof}[\tau]:(\textbf{var}[\tau]\to\textbf{comm})\to\textbf{exp}[\tau]$$

are suitable procedural constants, then

$$\#\textbf{new}[\tau]\iota.C$$

can be a conventional "block" containing a declaration of a local τ-valued variable, and

$$\#\textbf{valof}[\tau]\iota.C$$

can be a "block expression" whose value is the final value of the local variable. If $P:\theta$ and $\textbf{rec}[\theta]:(\theta\to\theta)\to\theta$ is a constant denoting the fixed-point operator for domain $[\![\theta]\!]$, then

$$\#\textbf{rec}[\theta]\iota.P$$

can be a "recursive labelling," much like the way LABEL was used in the original

version of LISP. We may then introduce a *recursive* form of definition:

recursive definition:

$$\frac{\begin{array}{cc}[\iota:\theta] & [\iota:\theta] \\ \vdots & \vdots \\ P:\theta & Q:\theta'\end{array}}{\textbf{letrec } \iota:\theta \textbf{ be } P \textbf{ in } Q:\theta'}$$

which may be de-sugared as follows:

$$\textbf{letrec } \iota:\theta \textbf{ be } P \textbf{ in } Q \equiv (\lambda\iota:\theta.Q)(\#\textbf{rec}[\theta]\iota.P).$$

A recursive variant of the procedure-definition form is also possible. It is convenient to introduce constants as follows:

undefined:

$$\frac{}{\textbf{undef}[\theta]:\theta}$$

with

$$\textbf{undef}[\theta] \equiv \#\,\textbf{rec}[\theta]\iota.\iota,$$

so that, for every u, $[\![\textbf{undef}[\theta]]\!]u = \perp_\theta$.

The quantification notation can also be used with *user-defined* "quantifiers"; for example, by defining a procedure

$$for\text{: } \textbf{exp[nat]} \times \textbf{exp[nat]} \to (\textbf{exp[nat]} \to \textbf{comm}) \to \textbf{comm}$$

recursively as follows:

$$for(m, n)(user)$$

$$= \textbf{if } m \leqslant n \textbf{ then } (user(m); for(\textbf{succ } m, n)(user)) \textbf{ else skip}$$

(or, for better efficiency, with a **while** loop), a programmer can then obtain a "for-loop" with local iteration index ι and body C as follows:

$$\#for(E_1, E_2)\iota.C,$$

where E_1 and E_2 are expressions for the iteration limits. Similar procedures can be defined to traverse trees, iterate over the elements of sets, etc. Several extensions to and generalizations of the quantification notation are described in [57].

4. Coercions

We have not yet explained how an acceptor or an expression can be obtained from a variable. Explicit selection operators could be provided, but it is conventional to use implicit conversions, or *coercions*. Our treatment is based on [35, 36, 44, 46, 49]. We adopt the principle that the applicability of a coercion from one type to another should not be dependent on syntactic context. Then, we can specify coercibility by

means of a binary relation $\vdash$ on phrase-type expressions. $\theta \vdash \theta'$ may be regarded as an abbreviation for the statement that, for any type assignment π and phrase Z, $\pi \vdash Z : \theta$ implies $\pi \vdash Z : \theta'$; i.e., in any context where phrases of type θ' are expected, any phrase of type θ can be used as well. (To prevent confusion, the turnstile symbol $\vdash$ will, from now on, be used only for coercibility relations.) It is natural to require reflexivity ($\theta \vdash \theta$) and transitivity ($\theta_1 \vdash \theta_2$ and $\theta_2 \vdash \theta_3$ imply $\theta_1 \vdash \theta_3$); i.e., $\vdash$ is a pre-order on phrase-type expressions.[2] The acceptor-selection and de-referencing coercions are then specified by requiring the following coercibility relationships:

$$\mathbf{var}[\tau] \vdash \mathbf{acc}[\tau], \qquad \mathbf{var}[\tau] \vdash \mathbf{exp}[\tau]$$

for every τ.

Semantically, we need to define a strict (i.e., $\perp$-preserving) continuous *conversion* function $[\![\theta \vdash \theta']\!] : [\![\theta]\!] \to [\![\theta']\!]$ whenever $\theta \vdash \theta'$. The conversions for acceptor-selection and de-referencing, $[\![\mathbf{var}[\tau] \vdash \mathbf{acc}[\tau]]\!]$ and $[\![\mathbf{var}[\tau] \vdash \mathbf{exp}[\tau]]\!]$, respectively, are the projections from $[\![\mathbf{var}[\tau]]\!] = [\![\mathbf{acc}[\tau]]\!] \times [\![\mathbf{exp}[\tau]]\!]$ to $[\![\mathbf{acc}[\tau]]\!]$ and $[\![\mathbf{exp}[\tau]]\!]$, respectively. To prevent ambiguity, we require that, for every θ, $[\![\theta \vdash \theta]\!]$ be the identity on $[\![\theta]\!]$, and, if $\theta_1 \vdash \theta_2$ and $\theta_2 \vdash \theta_3$, then $[\![\theta_1 \vdash \theta_3]\!]$ must be equal to the composite conversion $[\![\theta_1 \vdash \theta_2]\!]; [\![\theta_2 \vdash \theta_3]\!]$. In the language of category theory [17], this means that $[\![\cdot]\!]$ must be a *functor* from the pre-ordered set of phrase-type expressions (regarded as a category in the usual way) to the category of domains with least elements and strict continuous functions.

Similarly, we can define a pre-order $\vdash$ on the set of *data-type* expressions, with an interpretation function $[\![\cdot]\!]$ from this pre-ordered set to the usual category of sets and functions. We can now introduce new data types **int** and **real** with $[\![\mathbf{int}]\!]$ as the set of integers and $[\![\mathbf{real}]\!]$ as the set of real numbers, and then specify that $\mathbf{nat} \vdash \mathbf{int} \vdash \mathbf{real}$, with the obvious injections as the corresponding conversions.

We want any such $\tau \vdash \tau'$ to induce a coercion

$$\mathbf{exp}[\tau] \vdash \mathbf{exp}[\tau']$$

on phrase types. For example, an integer-valued expression should be usable wherever an expression producing reals is, because any integer value it produces can be converted to a real number. In general:

$$[\![\mathbf{exp}[\tau] \vdash \mathbf{exp}[\tau']]\!] e = e; [\![\tau \vdash \tau']\!]_\perp,$$

where $[\![\tau \vdash \tau']\!]_\perp$ is the $\perp$-preserving extension of $[\![\tau \vdash \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$ to $[\![\tau]\!]_\perp \to [\![\tau']\!]_\perp$. Furthermore, we want

$$\mathbf{acc}[\tau'] \vdash \mathbf{acc}[\tau]$$

that is to say,

$$\mathbf{exp}[\tau'] \to \mathbf{comm} \vdash \mathbf{exp}[\tau] \to \mathbf{comm}.$$

[2] Requiring anti-symmetry as well ($\theta \vdash \theta'$ and $\theta' \vdash \theta$ imply $\theta = \theta$) would make $\vdash$ a partial order, but this would be less convenient: it would be necessary to choose a canonical representative of each anti-symmetry equivalence class, and often this would be an arbitrary choice.

For example, an acceptor for reals should be usable wherever an acceptor for integers is, because any integer-valued expression to which it might be applied can first be converted to a real-valued expression.

In general, the coercibility of procedure types is determined by the coercibility of their argument and result types, as follows: if $\theta_0 \vdash \theta_0'$ and $\theta_1 \vdash \theta_1'$, then $\theta_0' \to \theta_1 \vdash \theta_0 \to \theta_1'$. Notice that the $\to$ operator is *anti-monotone* in its first operand. The conversion is defined by

$$[\![\theta_0' \to \theta_1 \vdash \theta_0 \to \theta_1']\!] f = [\![\theta_0 \vdash \theta_0']\!]; f; [\![\theta_1 \vdash \theta_1']\!],$$

where f is a function from $[\![\theta_0']\!]$ to $[\![\theta_1]\!]$.

It is convenient to add a new type **1** such that $\theta \vdash \mathbf{1}$ for *all* θ. Intuitively, **1** is the type of type-incorrect phrases and can be interpreted as denoting a singleton domain $\{\perp\}$; then $[\![\theta \vdash \mathbf{1}]\!]$ is the unique function from $[\![\theta]\!]$ to $[\![\mathbf{1}]\!]$, and $cond_{\mathbf{1}}(e, m_0, m_1) = \perp$. $[\![\mathbf{1}]\!]$ is a *terminal* object [17] in the category of domains. A complete program of type **1** should be considered as erroneous, and need not be executed since its semantics is trivial. But, in order that errors can be localized, we would expect a compiler to generate a message for any phrase of type **1** when none of its subphrases have type **1**; however, this must be only a *warning* message, because, in general, a complete program need not have type **1** when a proper component does. For example,

let x **be** P **in skip**

has type **comm**, even if $P : \mathbf{1}$. This simplifies the formal treatment of type-checking, and it is claimed in [50] that allowing well-formed programs to have ill-formed components is (part of) a solution to the syntactic problems discussed at the end of [43].

It is also convenient to adopt the coercion $\mathbf{1} \vdash \theta \to \mathbf{1}$ for all θ. To see why, suppose that the types of P and Q in a phrase of the form PQ cannot be "matched" as required by the syntax rule for procedure application. Then, by coercing P to type **1** and then to $\theta \to \mathbf{1}$ using the new coercion, such a match of types *is* possible for any type θ such that $Q : \theta$, and the resulting type of the application is **1**, as desired. $[\![\mathbf{1} \vdash \theta \to \mathbf{1}]\!]$ would be the function mapping the only element of **1** to the only element of $[\![\theta \to \mathbf{1}]\!]$.

In summary, we shall take the carriers for the pre-ordered sets of data and phrase-type expressions to be the smallest sets that contain all of the primitive type symbols discussed and, for phrase types, all of the finite composites freely generated by the $\to$ construction; furthermore, we shall take the $\vdash$ relations to be the smallest pre-orderings on these sets that include the specific relationships discussed and the rules of the preceding paragraphs for procedural types.

Our language has "generic" constructions, such as application, and for these it must be verified that the coercions do not lead to semantic ambiguities. For example, the rule for application is

$$\frac{P : \theta \to \theta' \quad Q : \theta}{PQ : \theta'}$$

and, if $P:\theta_1 \to \theta'$ and $Q:\theta_2, \theta$ can be *any* type satisfying $\theta_2 \vdash \theta \vdash \theta_1$; but our interpretations imply that

$$\begin{aligned}[\![PQ]\!]_{\pi\theta'}(u) &= [\![\theta_1 \to \theta' \vdash \theta \to \theta']\!](p)([\![\theta_2 \vdash \theta]\!]q) \\ &= [\![\theta' \vdash \theta']\!](p([\![\theta_2 \vdash \theta]\!]; [\![\theta \vdash \theta_1]\!])(q)) \\ &= p([\![\theta_2 \vdash \theta_1]\!]q),\end{aligned}$$

where $p = [\![P]\!]_{\pi(\theta_1 \to \theta')}(u)$ and $q = [\![Q]\!]_{\pi\theta_2}(u)$. The last line is independent of θ, so that the syntactic ambiguity does not create a semantic ambiguity. Similar results can be proved [3, 44, 49] about the interpretations of $:=$, $=$, $+$, and so on (ignoring overflow and roundoff errors), and also for syntactic sugarings that involve type inference, such as **let**-definition and quantification.

The conditional construction, however, is more problematical. The syntax rule is

$$\frac{B:\mathbf{exp[bool]} \quad Z_1:\theta \quad Z_2:\theta}{\mathbf{if}\ B\ \mathbf{then}\ Z_1\ \mathbf{else}\ Z_2:\theta}$$

and this means that, if $Z_1:\theta_1$ and $Z_2:\theta_2$, θ should be an upper bound of θ_1 and θ_2, where θ' is regarded as *greater* than θ when $\theta \vdash \theta'$. To ensure that a conditional phrase has a most-general type (up to anti-symmetry), every such θ_1 and θ_2 should have a *least* upper bound. The problem is that at present there are pairs of types that do not have least upper bounds; for example, **var[int]** and **var[real]** have **exp[real]** and **acc[int]** as upper bounds, but no *least* upper bound. But then the construct **if** B **then** Z_1 **else** Z_2 is not well-formed when Z_1:**var[int]** and Z_2:**var[real]**, and clearly this is unacceptable.

A solution to this problem [44, 46] is to augment the type structure to allow for variables whose two components accept and produce values of different types. If θ denoted the type of integer-accepting and real-producing variables, then θ would be a least upper bound of **var[int]** and **var[real]**. The conditional construct above would then be well-formed and usable as the left-hand side of an integer assignment and as the right-hand side of a real assignment. This will turn out to be just one application of a more general extension [50] to be discussed in the following section.

5. Conjunctive types

We now add to the type structure of our language a product-like operation on types which was originally studied [10] in the context of type assignment for *untyped* languages and has recently [50] been adapted to *typed* languages:

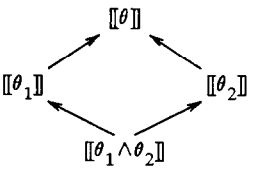
$$\theta ::= \cdots | \theta_1 \wedge \theta_2.$$

The key idea is that the conjunctive type $\theta_1 \wedge \theta_2$ must support the following "selection" coercions:

$$\theta_1 \wedge \theta_2 \vdash \theta_i \quad \text{for } i = 1, 2.$$

For example, we can now *define* **var**[τ] to be **acc**[τ] $\wedge$ **exp**[τ] (rather than a primitive type); the acceptor-selection and de-referencing coercions turn out to be selection coercions on conjunctive types. Furthermore, elements of type **acc**[**int**] $\wedge$ **exp**[**real**] are the "mixed" variables needed to allow **if** B **then** Z_1 **else** Z_2 when Z_1 : **var**[**int**] and Z_2 : **var**[**real**]. We will describe additional applications of conjunctive types to generic procedures and to the definition of products (records) after discussing their semantics.

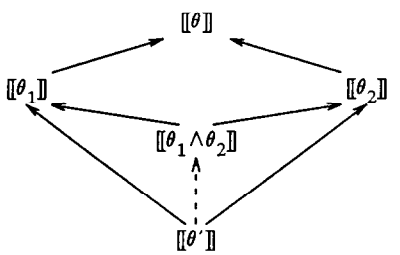
In general, $\wedge$ cannot be interpreted as a conventional product, because, if θ is any upper bound of θ_1 and θ_2, the following diagram must commute:



where all of the arrows are conversions. If θ_1 and θ_2 have a least upper bound θ, the appropriate general interpretation is as follows:

$$[\![\theta_1 \wedge \theta_2]\!] = \{\langle a, b\rangle \in [\![\theta_1]\!] \times [\![\theta_2]\!] \mid [\![\theta_1 \vdash \theta]\!] a = [\![\theta_2 \vdash \theta]\!] b\},$$

ordered component-wise, and the selection conversions $[\![\theta_1 \wedge \theta_2 \vdash \theta_i]\!]$ are the obvious restrictions of the projection functions from $[\![\theta_1]\!] \times [\![\theta_2]\!]$. The interpretation of $\theta_1 \wedge \theta_2$ is termed [17] the *pullback* of $[\![\theta_1 \vdash \theta]\!]$ and $[\![\theta_2 \vdash \theta]\!]$, and it can be proved that, for any θ' that is a lower bound of θ_1 and θ_2, there is a unique conversion $[\![\theta' \vdash \theta_1 \wedge \theta_2]\!]$ for which the following diagram of conversions commutes:



Note that the requirement that all conversions be *strict* ensures that $[\![\theta_1 \wedge \theta_2]\!]$ has a least element. We may also define

$$cond_{\theta_1 \wedge \theta_2}(e, \langle m_1, m_2\rangle, \langle m_1', m_2'\rangle) = \langle cond_{\theta_1}(e, m_1, m_1'), cond_{\theta_2}(e, m_2, m_2')\rangle.$$

We may now complete the definition of the particular pre-ordered set of phrase-type expressions that results from adding conjunctive type expressions to our language. In addition to the coercions discussed earlier, we want $\theta_1 \wedge \theta_2$ to be the greatest lowest bound of θ_1 and θ_2:

if $\theta \vdash \theta_1$, $\theta \vdash \theta_2$ then $\theta \vdash \theta_1 \wedge \theta_2$.

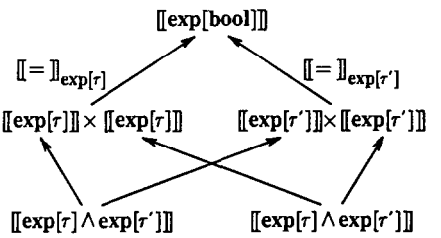
It follows that, modulo anti-symmetry, the operation $\wedge$ has identity **1** and is idempotent, commutative and associative. For example, the fragment of Θ relevant

for integer and/or real acceptors, expressions, and variables is diagrammed in Fig. 1. Finally, the treatment of procedures is simplified by adopting the following coercion:

$$(\theta \to \theta_1) \wedge (\theta \to \theta_2) \vdash \theta \to (\theta_1 \wedge \theta_2),$$

which is a generalization of the coercion from **1** to $\theta \to \mathbf{1}$. The interpretations of these coercions should be clear.

The introduction of conjunctive types creates a problem for operators that are generic over data types, such as the equality test. Suppose that the operands have conjunctive type $\mathbf{exp}[\tau] \wedge \mathbf{exp}[\tau']$; then the following diagram illustrates the ambiguity possible:



The unlabelled arrows at the bottom of the diagram are the selections from the conjunctive types. The problem is that equality of the τ-valued components may not correspond to equality of the τ'-valued components. Fortunately, we do not need phrases having such conjunctive types, and so the problem is only a technical one. Solutions that can be adopted include

- making $\wedge$ a *partial* operation, so the undesired conjunctive types do not exist;
- introducing a "dummy" upper bound of all the data types, interpreted as the union of all the data types, so that $\mathbf{exp}[\tau] \wedge \mathbf{exp}[\tau']$ would then be interpreted as the domain of functions from states to the *intersection*, and not the product, of τ and τ';
- adopting a nonuniform interpretation of type conjunction.

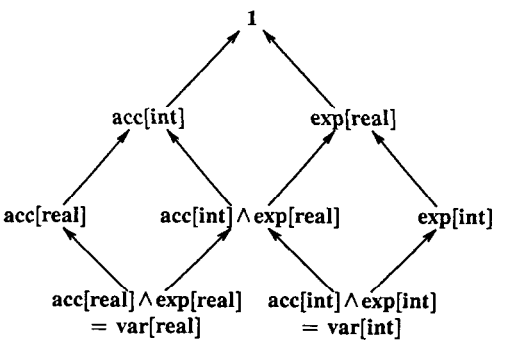


Fig. 1.

The subsequent development is not affected significantly by the choice of solution.

We can now consider how programmers might define values of conjunctive type. The problem is to ensure that the commutativity constraints are not violated. In [50], it is proposed that the abstraction notation be generalized to allow definition of generic procedures:

generic abstraction:

$$\frac{\begin{array}{l}[\iota : \theta_i] \\ \quad\vdots \qquad \text{for } i = 1, 2, \ldots, n \\ P : \theta_i'\end{array}}{\lambda\iota : \theta_1, \ldots, \theta_n . P : (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n')}$$

When $n > 1$, this provides a way to define generic procedures; for example,

$$\lambda i : \mathbf{exp[nat], exp[int], exp[real]} . i + i$$

denotes a generic "doubling" function of type

$$(\mathbf{exp[nat]} \to \mathbf{exp[nat]})$$
$$\wedge\, (\mathbf{exp[int]} \to \mathbf{exp[int]})$$
$$\wedge\, (\mathbf{exp[real]} \to \mathbf{exp[real]}).$$

The interpretation of generic abstraction is

$$[\![\lambda\iota : \theta_1, \ldots, \theta_n . P]\!] u = \langle f_1, \ldots, f_n \rangle,$$
$$\text{where } f_i(a) = [\![P]\!](u \,|\, \iota \mapsto a) \quad \text{for all } a \in [\![\theta_i]\!],\ 1 \leqslant i \leqslant n$$

and it can be shown [50] that the components of the n-tuple of functions satisfy the commutativity property

$$[\![\theta_i \to \theta_i' \vdash \theta]\!] f_i = [\![\theta_j \to \theta_j' \vdash \theta]\!] f_j, \quad 1 \leqslant i, j \leqslant n$$

for any θ that is an upper bound of all the $\theta_i \to \theta_i'$. This means that generic abstraction is not merely "overloading" as found in, say, PL/I; the components of the generic meaning satisfy a commutatitivity constraint that ensures that there can be no semantic ambiguity, even in the presence of coercions.

It is also possible to extend the procedure-definition notation to allow definition of such generic procedures; for simplicity, the following rule allows procedures with one parameter only, but the generalization to multiple parameters is obvious;

generic procedure definition:

$$\frac{\begin{array}{ll}[\iota' : \theta_i] & [\iota : (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n')] \\ \quad\vdots \quad \text{for } i = 1, 2, \ldots, n & \qquad\vdots \\ P : \theta_i' & \qquad Q : \theta'\end{array}}{\textbf{let } \iota : (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n') \textbf{ in } Q \textbf{ where } \iota(\iota') = P : \theta'}$$

The de-sugaring is

$$\textbf{let } \iota : (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n') \textbf{ in } Q \textbf{ where } \iota(\iota') = P$$
$$\equiv (\lambda\iota : (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n').\, Q)(\lambda\iota' : \theta_1, \ldots, \theta_n.\, P).$$

Reynolds [50] also proposes a form of merging for procedures. This can be slightly generalized [3] as follows:

procedure merging:

$$\frac{\begin{array}{cc} & [\iota' : \theta_i] \\ & \vdots \qquad \text{for } i = 1, 2, \ldots, n \\ P : \theta_0 \quad Q : \theta_i' & \end{array}}{P, \lambda\iota : \theta_1, \ldots, \theta_n.\, Q : \theta_0 \wedge (\theta_1 \to \theta_1') \wedge \cdots \wedge (\theta_n \to \theta_n')}$$

provided that, if $\theta_0 \vdash \theta \to \theta'$, then $\mathbf{1} \vdash \theta'$ or θ is not conjoinable with any of the θ_i. The restriction on θ_0 ensures that there are no commutativity constraints on the value of P so that the conjunction $\theta_0 \wedge \cdots$ is interpreted as a full product. For example, this construction allows a programmer to define a nonstandard "variable" by merging a (suitably-typed) acceptor procedure with an expression, and to merge a Boolean negation procedure of type **exp[bool]** → **exp[bool]** with a numerical negation of type (**exp[int]** → **exp[int]**) ∧ (**exp[real]** → **exp[real]**). The latter is possible because the type **exp[bool]** ∧ **exp[int]** ∧ **exp[real]** is disallowed (or trivial).

6. Phrase-type structures

In this section we begin (at last) to discuss facilities for creating and using composite objects such as records and arrays. Because we have two kinds of types, phrase types and data types, we will consider two kinds of structures. *Phrase-type* structures will be denotable by identifiers (but not expressible or assignable to variables).

6.1. Products

To construct products (of phrase types), we first introduce *tagged* types, as follows:

$$\theta ::= \cdots \mid \iota \mapsto \theta$$

where ι here stands for an arbitrary identifier. We could define $[\![\iota \mapsto \theta]\!]$ to be just $[\![\theta]\!]$, but, for clarity, we let $[\![\iota \mapsto \theta]\!]$ be the isomorphic domain of functions $\{\iota\} \to [\![\theta]\!]$. We then introduce the following coercions:

$$\text{if } \theta \vdash \theta' \text{ then } \iota \mapsto \theta \vdash \iota \mapsto \theta',$$
$$\iota \mapsto \theta_1 \wedge \iota \mapsto \theta_2 \vdash \iota \mapsto (\theta_1 \wedge \theta_2),$$
$$\mathbf{1} \vdash \iota \mapsto \mathbf{1},$$

denoting the following conversions:

$$[\![\iota \mapsto \theta \vdash \iota \mapsto \theta']\!] f = f; [\![\theta \vdash \theta']\!],$$

$$[\![\iota \mapsto \theta_1 \wedge \iota \mapsto \theta_2 \vdash \iota \mapsto (\theta_1 \wedge \theta_2)]\!] \langle f_1, f_2 \rangle (\iota) = \langle f_1(\iota), f_2(\iota) \rangle,$$

$$[\![\mathbf{1} \vdash \iota \mapsto \mathbf{1}]\!] x\iota = x,$$

and define

$$cond_{\iota \mapsto \theta}(b, f_0, f_1)(\iota) = cond_\theta(b, f_0(\iota), f_1(\iota)).$$

Following [50], we now provide notation for defining "tagged" meanings:

tag introduction

$$\frac{P : \theta_0 \quad Q : \theta_1}{P, \iota \mapsto Q : \theta_0 \wedge \iota \mapsto \theta_1}$$

provided $\theta_0 \vdash \iota \mapsto \theta$ only if $\mathbf{1} \vdash \theta$. The constraint ensures that the ι-tagged meaning of Q *replaces* any similarly-tagged value derivable from P. The interpretation is

$$[\![P, \iota \mapsto Q]\!] u = \langle [\![P]\!] u, f \rangle, \quad \text{where } f(\iota) = [\![Q]\!] u.$$

Note that the occurrence of ι as a tag in $\iota \mapsto Q$ is not a conventional identifier occurrence; for example, it is not subject to substitutions.

Tag introduction can be used to create arbitrary n-tuples of tagged meanings. We can allow

$$\iota_1 \mapsto P_1, \ldots, \iota_n \mapsto P_n$$

as an abbreviation for

$$\mathbf{undef}[\mathbf{1}], \iota_1 \mapsto P_1, \ldots, \iota_n \mapsto P_n.$$

A particular "field" of such a "record" or "structure" can then be selected by the selection coercion for conjunctive types, and the tag can be removed by using:

tag elimination:

$$\frac{Q : \iota \mapsto \theta}{Q.\iota : \theta}$$

with the following interpretation:

$$[\![Q.\iota]\!] u = [\![Q]\!](u)(\iota).$$

Note that the combination of conjunctive and tagged types gives essentially the same power as conventional "named-field" products, including the coercions of [46], which have the "inheritance" properties discussed in [9].

As an example of how phrase-type products can be used, let **class**[θ] be an abbreviation for ($\theta \to$ **comm**) $\to$ **comm**; for example, the variable-declaration quantifier **new**[τ] is of phrase type **class**[**var**[τ]]. Then the following program fragment illustrates how to define and create an instance of an abstract "class" of "counter" objects, whose capabilities are limited to incrementation and evaluation:

let *counter* : **class**[*inc* $\mapsto$ **comm** $\wedge$ *val* $\mapsto$ **exp**[**nat**]]
in :
($\#$ *counter c*.
$\cdots$ *c.inc* $\cdots$ *c.val* $\cdots$)
:
where *counter*(*user*)
$=$ $\#$**new**[**nat**] *n*. *n* := **0**;
user(*inc* $\mapsto$ *n* := *n* + **1**, *val* $\mapsto$ *n*).

The procedural parameter *user* may be thought of as a typical "customer" for an instance of the counter class. The representation of a counter (i.e., variable *n*) is a "private" variable, not directly accessible to users. In more complex examples, private *procedures* might be necessary. Note that *counter*, as defined in this example, is a *value*, not a type; it can be used as a quantifier to create counter objects, but cannot be used, for example, as the type of a parameter.

6.2. Sums

Typed programming languages generally have "union" or "variant" mechanisms to allow strict type constraints to be relaxed in a disciplined way. We therefore consider *disjunctive* phrase types as follows

$$\theta ::= \cdots \mid \theta_1 \vee \theta_2$$

and the following coercions, which are essentially dual to those for conjunctive types:

$\theta_i \vdash \theta_1 \vee \theta_2$ for $i = 1, 2$,

if $\theta_1 \vdash \theta$, $\theta_2 \vdash \theta$ then $\theta_1 \vee \theta_2 \vdash \theta$.

We defer consideration of the semantics of these mechanisms.

Now, to form "sums" (which are like disjoint unions), we will want to form disjunctions of types tagged with distinct tags. However, we cannot use, for example, $\iota_1 \mapsto \theta_1 \vee \iota_2 \mapsto \theta_2$, because then $\iota_1 \mapsto \theta_1$ and $\iota_2 \mapsto \theta_2$ would have a nontrivial upper bound and the pullback interpretation of $\iota_1 \mapsto \theta_1 \wedge \iota_2 \mapsto \theta_2$ would no longer yield a full product. Hence, we introduce a second form of tagged type, as follows:

$$\theta ::= \cdots \mid \iota \mathbin{@} \theta,$$

with coercion

if $\theta \vdash \theta'$ then $\iota \mathbin{@} \theta \vdash \iota \mathbin{@} \theta'$.

We can then form "sum" types $\iota_1 \mathbin{@} \theta_1 \vee \cdots \vee \iota_n \mathbin{@} \theta_n$, and define values of such types

by using the following syntax rule:

sum introduction:

$$\frac{P:\theta}{\iota @ P:\iota @ \theta}$$

and the coercion from $\iota @ \theta$ into $\cdots \vee \iota @ \theta \vee \cdots$.

To avoid introducing undesired upper bounds, we make disjunction a *partial* operation, applicable only to @-tagged types and disjunctions of them (except for trivial disjunctions such as $\theta \vee \theta$). Hence, we introduce a binary relation ∇ on phrase-type expressions, termed *disjoinability*, and deem $\theta_1 \vee \theta_2$ to be well-formed only when $\theta_1 \nabla \theta_2$. We define ∇ to be the smallest reflexive and symmetric binary relation on phrase-type names that has the following properties:

if $\theta \nabla \theta_i$ for all $1 \leqslant i \leqslant n$ then $\theta \nabla (\theta_1 \vee \cdots \vee \theta_n)$

if $\iota = \iota'$ implies $\theta \nabla \theta'$ then $\iota @ \theta \nabla \iota' @ \theta'$.

Although sum types are discussed in [46], their interpretation is rather problematical. We can dispose of trivial uses of disjunction as follows:

$$[\![\theta \vee \theta]\!] = [\![\theta]\!],$$

$$[\![\iota @ \theta_1 \vee \iota @ \theta_2]\!] = [\![\iota @ (\theta_1 \vee \theta_2)]\!],$$

because of the coercions $\theta \vee \theta \vdash \theta$ and $\iota @ \theta_1 \vee \iota @ \theta_2 \vdash \iota @ (\theta_1 \vee \theta_2)$. This leaves us sums of the form $\iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n$ to interpret, where the ι_i are distinct. The category of directed-complete partial orders and continuous functions has co-products of domains D_i, $i \in I$, as follows:

$$\sum_{i \in I} D_i = \bigcup_{i \in I} \{\langle i, v\rangle \mid v \in D_i\},$$

ordered by $\langle i, v_i\rangle \sqsubseteq \langle j, v_j\rangle$ iff $i = j$ and $v_i \sqsubseteq v_j$. But such domains do not have *least* elements. Lifting a co-product to introduce a least element produces the so-called "separated sum" of domains (with least elements):

$$[\![\iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n]\!] = \left(\sum_{1 \leqslant i \leqslant n} [\![\theta_i]\!] \right)_\perp$$

and then

$$[\![\iota @ P]\!] us = \langle 1, [\![P]\!] u\rangle.$$

Unfortunately, there is no apparent way to define a function $cond_\theta$ when θ is a sum type. This not only violates the principle of uniformity, but also, and more importantly, precludes the use of conditional phrases to define values of sum types. Our solution to the latter problem is to introduce a new phrase type of "pure" (i.e., state-independent) value phrases, as follows

$$\theta ::= \cdots \mid \mathbf{var}[\tau],$$

with $[\![\mathbf{var}[\tau]]\!] = [\![\tau]\!]_\perp$, and then $[\![\mathbf{exp}[\tau]]\!] = S \to [\![\mathbf{var}[\tau]]\!]$. Literal constants such as **0** and **true** can now be treated as having phrase types **val**[**nat**] and **val**[**bool**], respectively, and, to allow these to be used in expressions, we introduce a coercion $\mathbf{val}[\tau] \vdash \mathbf{exp}[\tau]$ with interpretation $[\![\mathbf{val}[\tau] \vdash \mathbf{exp}[\tau]]\!] vs = v$, for all states s. We can also allow expression operators such as + to be used with *value* phrases by requiring the following kind of commutativity:

$$\begin{array}{ccc} [\![\mathbf{exp}[\tau]]\!] \times [\![\mathbf{exp}[\tau]]\!] & \xrightarrow{[\![+]\!]_{\mathbf{exp}[\tau]}} & [\![\mathbf{exp}[\tau]]\!] \\ \uparrow \quad \uparrow & & \uparrow \\ [\![\mathbf{val}[\tau]]\!] \times [\![\mathbf{val}[\tau]]\!] & \xrightarrow[[\![+]\!]_{\mathbf{val}[\tau]}]{} & [\![\mathbf{val}[\tau]]\!] \end{array}$$

and similarly for all of the other operators on expressions, such as $=$, **not**, **and**, $<$, **succ**, etc.

Of course, we cannot define a function $cond_\theta$ when θ is a value type; however, we can extend the applicability of the **if** construction as follows:

value conditional:

$$\frac{V : \mathbf{val}[\mathbf{bool}] \quad Z_0 : \theta \quad Z_1 : \theta}{\mathbf{if}\ V\ \mathbf{then}\ Z_0\ \mathbf{else}\ Z_1 : \theta}.$$

The condition V has type **val**[**bool**] rather than **exp**[**bool**]. The interpretation, for *every* phrase type θ (including value and sum types), is

$$[\![\mathbf{if}\ V\ \mathbf{then}\ Z_0\ \mathbf{else}\ Z_1]\!]u = \begin{cases} [\![Z_0]\!]u, & \text{if } [\![V]\!]u = \mathit{true}, \\ [\![Z_1]\!]u, & \text{if } [\![V]\!]u = \mathit{false}, \\ \perp_\theta, & \text{if } [\![V]\!]u = \perp. \end{cases}$$

We now consider how to interpret the coercions. Apart from trivial coercions such as $\theta \vee \theta \vdash \theta$ and $\iota @ \theta_1 \vee \iota @ \theta_2 \vdash \iota @ (\theta_1 \vee \theta_2)$, coercions on disjunctive types have the general form

$$\text{if } \{\iota_1, \ldots, \iota_n\} \subseteq \{\iota'_1, \ldots, \iota'_m\} \text{ and } \theta_i \vdash \theta'_j \text{ whenever } \iota_i = \iota'_j \text{ then}$$

$$\iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n \vdash \iota'_1 @ \theta'_1 \vee \cdots \vee \iota'_m @ \theta'_m,$$

where the ι_i are distinct and the ι'_j are distinct; such coercions can be interpreted as follows:

$$[\![\iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n \vdash \iota'_1 @ \theta'_1 \vee \cdots \vee \iota'_m @ \theta'_m]\!]z$$

$$= \begin{cases} \langle j, [\![\theta_i \vdash \theta'_j]\!]v \rangle, & \text{if } z = \langle i, v \rangle \text{ and } \iota_i = \iota'_j, \\ \perp, & \text{if } z = \perp. \end{cases}$$

To discriminate among the variants possible for a value of sum type, we introduce notation for

sum elimination:

$$\frac{Z : \iota_i @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n}{\mathbf{casetag}[\theta]Z : (\iota_1 \mapsto (\theta_1 \to \theta) \wedge \cdots \wedge \iota_n \mapsto (\theta_n \to \theta)) \to \theta}$$

where here, and throughout the rest of this section, the ι_i are distinct. The interpretation is as follows:

$$[\![\mathbf{casetag}[\theta]Z]\!]u\langle \ldots, f_i, \ldots\rangle = \begin{cases} \bot, & \text{if } [\![Z]\!]u = \bot, \\ f_i(v), & \text{if } [\![Z]\!]u = \langle i, v\rangle. \end{cases}$$

Then, if we adopt the following form of

distributed quantification:

$$\frac{Q : (\iota_1 \mapsto (\theta_1 \to \theta_1') \wedge \cdots \wedge \iota_n \mapsto (\theta_n \to \theta_n')) \to \theta \qquad \begin{array}{c}[\iota : \theta_i] \\ \vdots \\ P_i : \theta_i'\end{array} \text{ for } i = 1, 2, \ldots, n}{\#Q\iota.\iota_1 \mapsto P_1 | \cdots | \iota_n \mapsto P_n : \theta}$$

with de-sugaring

$$\#Q\iota.\iota_1 \mapsto P_1 | \cdots | \iota_n \mapsto P_n$$
$$\equiv Q(\iota_n \mapsto (\lambda\iota : \theta_1 . P_1), \ldots, \iota_n \mapsto (\lambda\iota : \theta_n . P_n)),$$

we have the following as a *derived* syntax rule:

$$\frac{Z : \iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n \qquad \begin{array}{c}[\iota : \theta_i] \\ \vdots \\ P_i : \theta\end{array} \text{ for } i = 1, 2, \ldots, n}{\#(\mathbf{casetag}[\theta]Z)\iota.\iota_1 \mapsto P_1 | \cdots | \iota_n \mapsto P_n : \theta}$$

The tag on the value of Z is used to select one of the P_i; free occurrences of ι in that P_i denote the value of Z with the tag removed.

It is not clear how useful sums of (phrase) types will be, particularly when conditional phrases of sum types must be constrained to purely-applicative contexts. A realistic class of applications would appear to be recursively-defined types with "lazy" components, as discussed in [7, 8, 16].

It is also possible to introduce a "bottom" type **0**, dual to **1**:

$$\theta ::= \cdots | \mathbf{0}$$

with coercion

$$\mathbf{0} \vdash \theta$$

for any θ, and interpretations

$$[\![\mathbf{0}]\!] = \{\bot\},$$
$$[\![\mathbf{0} \vdash \theta]\!]z = \bot_\theta.$$

The bottom type can be made disjoinable with every type, because $\mathbf{0} \vee \theta \vdash \theta$. The bottom type is not particularly useful, but it does allow the *family* of constants **under**$[\theta]$ for every θ to be replaced by a *single* constant **undef** of type **0**, because its value converts to the least element of any phrase type.

6.3. Arrays

Arrays differ from products in two ways: components of arrays can be selected by using a computed "index" (rather than a static "tag"), and, to make this practical, all of the components of an array must have the same type.

Programming languages typically provide arrays in two specialized and unrelated forms: as conventional arrays of variables and in "case" constructions, which involve indexing into what are essentially arrays of commands or expressions. Furthermore, the difference between arrays and procedures is essentially a question of representation. To generalize and unify all of these possibilities, we use procedures as arrays.

One can, for example, define a class-returning procedure

$$\mathit{NewRealVarArray} : \mathbf{exp[nat]} \to \mathbf{class}[(\mathbf{exp[nat]} \to \mathbf{var[real]}) \wedge \mathit{size} \mapsto \mathbf{exp[nat]}]$$

as follows:

$$\begin{aligned}
&\mathit{NewRealVarArray}(\mathit{size})(\mathit{user})\\
&= \mathbf{let}\ \mathit{allocate}\colon \mathbf{exp[nat]} \times (\mathbf{exp[nat]} \to \mathbf{var[real]}) \to \mathbf{comm}\\
&\quad \mathbf{in}\ \mathit{allocate}(\mathbf{0}, \mathbf{undef})\\
&\quad \mathbf{whererec}\ \mathit{allocate}(i, a)\\
&\qquad = \mathbf{if}\ i = \mathit{size}\ \mathbf{then}\ \mathit{user}(a, \mathit{size} \mapsto i)\ \mathbf{else}\\
&\qquad\quad (\#\,\mathbf{new[real]}x.\\
&\qquad\qquad \mathbf{let}\ \mathit{newa}\colon \mathbf{exp[nat]} \to \mathbf{var[real]}\\
&\qquad\qquad \mathbf{in}\ \mathit{allocate}(\mathbf{succ}\ i, \mathit{newa})\\
&\qquad\qquad \mathbf{where}\ \mathit{newa}(j) = \mathbf{if}\ i = j\ \mathbf{then}\ x\ \mathbf{else}\ a(j)).
\end{aligned}$$

Then,

$$\#\,\mathit{NewRealVarArray}(n)\ A.\cdots A(i)\cdots A.\mathit{size}\cdots$$

declares A to be an "array" of n real variables, which also can be asked its "size." The possible indices are the natural numbers less than n. Each call of *allocate* (except the last) declares one real variable; the last call of *allocate* (when i reaches *size*) applies the *user* procedure to the array of variables, represented by a procedure. In practice, an equivalent but more efficient implementation would be provided as the value of a constant or pre-defined identifier.

To allow a convenient way to construct arrays with "randomly-accessible" components, we introduce a class of static set expressions as follows:

$$\sigma ::= \nu \mid \nu..\nu' \mid \sigma, \sigma'$$

where ν ranges over numerals (or, more generally, statistically-evaluable numerical expressions). If $[\![\nu]\!]_{\mathbf{nat}}$ is the natural number denoted by ν, these forms of set expression are interpreted as follows:

$$[\![\nu]\!] = \{[\![\nu]\!]_{\mathbf{nat}}\},$$

$$[\![\nu..\nu']\!] = \{i \in [\![\mathbf{nat}]\!] \mid [\![\nu]\!]_{\mathbf{nat}} \leqslant i \leqslant [\![\nu']\!]_{\mathbf{nat}}\},$$

$$[\![\sigma, \sigma']\!] = [\![\sigma]\!] \cup [\![\sigma']\!].$$

Then "arrays" may be created by using the following syntax:

array construction:

$$\frac{P : \mathbf{exp}[\mathbf{nat}] \to \theta \quad Q : \theta}{P\,|\,[\sigma]Q : \mathbf{exp}[\mathbf{nat}] \to \theta}$$

with interpretation

$$[\![P\,|\,[\sigma]Q]\!]ue = cond_\theta(b, [\![Q]\!]u, [\![P]\!]ue),$$

where $b(s) = (e(s) \in [\![\sigma]\!])$, provided that θ is a type for which a function $cond_\theta$ is definable; otherwise (i.e., for value types, sum types, and procedural, tagged, and conjunctive types constructed from these), it is necessary to use

value-indexed array construction:

$$\frac{P : \mathbf{val}[\mathbf{nat}] \to \theta \quad Q : \theta}{P\,|\,[\sigma]Q : \mathbf{val}[\mathbf{nat}] \to \theta}$$

with interpretation

$$[\![P\,|\,[\sigma]Q]\!]uv = \begin{cases} \perp, & \text{if } v = \perp, \\ [\![Q]\!]u, & \text{if } v \in [\![\sigma]\!], \\ [\![P]\!]uv, & \text{otherwise.} \end{cases}$$

We can allow

$$[\sigma_1]P_1\,|\cdots|\,[\sigma_n]P_n$$

as an abbreviation for

$$\mathbf{undef}\,|\,[\sigma_1]P_1\,|\cdots|\,[\sigma_n]P_n.$$

For example, a multi-way conditional-selection construct may be obtained by regarding **case** E **of** P as an alternative syntactic form of the procedure application $P(E)$. If P has the form

$$[\nu_1]P_1\,|\cdots|\,[\nu_n]P_n,$$

the effect is as expected: the value of E is used to select the appropriate P_i; if the value of E does not correspond to any of the numerals, the result is undefined (i.e., $\perp$).

6.4. Axioms

In this subsection, we state axioms which are validated by the semantics given in the preceding subsections for the features supporting phrase-type structuring.

The axioms appropriate to tag introduction and elimination are as follows:

$$(P, \iota \mapsto Q).\iota \equiv Q,$$

$$(P, \iota \mapsto Q).\iota' \equiv P.\iota'$$

for $\iota' \neq \iota$. We can then derive that for all P_i and distinct ι_i,

$$(\iota_1 \mapsto P_1, \ldots, \iota_n \mapsto P_n).\iota_i \equiv P_i.$$

We also have that for $\theta = \iota_1 \mapsto \theta_1 \wedge \cdots \wedge \iota_n \mapsto \theta_n$ and any $P : \theta$,

$$(\iota_1 \mapsto P.\iota_1, \ldots, \iota_n \mapsto P.\iota_n) \equiv P,$$

and for B : **exp**[**bool**] and $P_1, P_2 : \iota \mapsto \theta$,

$$(\textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2).\iota \equiv \textbf{if } B \textbf{ then } P_1.\iota \textbf{ else } P_2.\iota,$$

and similarly if B : **val**[**bool**].

The axioms for *sums* of phrase types are as follows:

$$\#(\textbf{casetag}[\theta]\iota_i @ P)\iota.\cdots|\iota_i \mapsto P_i|\cdots \equiv \textbf{let } \iota \textbf{ be } P \textbf{ in } P_i,$$

for $\theta = \iota_1 @ \theta_1 \vee \cdots \vee \iota_n @ \theta_n$ and any $Z : \theta$,

$$\#(\textbf{casetag}[\theta]Z)\iota.\iota_1 \mapsto (\iota_1 @ \iota)|\cdots|\iota_n \mapsto (\iota_n @ \iota) \equiv Z;$$

and, for V : **val**[**bool**],

$$\textbf{casetag}[\theta](\textbf{if } V \textbf{ then } Z_1 \textbf{ else } Z_2)$$

$$\equiv \textbf{if } V \textbf{ then } (\textbf{casetag}[\theta]\ Z_1) \textbf{ else } (\textbf{casetag}[\theta]\ Z_2).$$

Note that the equivalence

$$\iota @ (\textbf{if } V \textbf{ then } P_1 \textbf{ else } P_2) \equiv \textbf{if } V \textbf{ then } \iota @ P_1 \textbf{ else } \iota @ P_2$$

fails, but only when $[\![V]\!]u = \perp$.

Finally, for array construction, we have the axiom

$$P|[\nu_1]P_1|\cdots|[\nu_n]P_n$$

$$\equiv \lambda\iota : \theta.\textbf{if } \iota = \nu_n \textbf{ then } P_n$$

$$\textbf{else if } \iota = \nu_{n-1} \textbf{ then } P_{n-1}$$

$$\vdots$$

$$\textbf{else if } \iota = \nu_1 \textbf{ then } P_1$$

$$\textbf{else } P(\iota)$$

where ι is not free in the P_i, and θ, the type of ι, is **exp[nat]** if possible, and otherwise **val[nat]**.

7. Data-type structures

Although much can be done with only phrase-type structures [50], the author believes that data-type structures will also be necessary in practice. Elements of data types are expressible and assignable to appropriately-typed variables, and it is straightforward to create new named variables for such values, using **new**$[\tau]$ quantifiers. In this section, we consider products, sums, and arrays of *data* types. One of the main aims of our design is to ensure as much conceptual and notational compatibility between corresponding data-type and phrase-type structures as possible. Unfortunately, this compatibility does not allow us to simplify the *formal* descriptions of the facilities.

7.1. Data-type products

There seems to be no reason to provide conjunctions of data types other than to allow products, so that we extend the language of data-type expressions as follows:

$$\tau ::= \cdots \mid \iota \mapsto \tau \mid \mathbf{1} \mid \tau_1 \wedge \tau_2, \quad (\text{when } \tau_1 \triangle \tau_2),$$

where $\triangle$ (data-type conjoinability) is the smallest reflexive and symmetric binary relation on data-type expressions that satisfies

$$\text{if } \iota = \iota' \text{ implies } \tau \triangle \tau' \text{ then } \iota \mapsto \tau \triangle \iota' \mapsto \tau',$$

$$\text{if } \tau \triangle \tau_i \text{ for all } 1 \leqslant i \leqslant n \text{ then } \tau \triangle (\tau_1 \wedge \cdots \wedge \tau_n).$$

The appropriate coercions are

$$\text{if } \tau \vdash \tau' \text{ then } \iota \mapsto \tau \vdash \iota \mapsto \tau',$$

$$\iota \mapsto \tau \wedge \iota \mapsto \tau' \vdash \iota \mapsto (\tau \wedge \tau'),$$

$$\tau \vdash \mathbf{1},$$

$$\mathbf{1} \vdash \iota \mapsto \mathbf{1},$$

$$\tau_1 \wedge \tau_2 \vdash \tau_i \quad \text{for } i = 1, 2,$$

$$\text{if } \tau \vdash \tau_1, \tau \vdash \tau_2 \text{ then } \tau \vdash \tau_1 \wedge \tau_2.$$

The nontrivial parts of the interpretation of the new data-type names are as follows:

$$[\![\mathbf{1}]\!] = \{*\} \quad (\text{any singleton set}),$$

$$[\![\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n]\!] = \prod_{1 \leqslant i \leqslant n} [\![\tau_i]\!],$$

where the ι_i are distinct. As in [9, 46], nontrivial coercions have the form

if $\{\iota'_1, \ldots, \iota'_m\} \subseteq \{\iota_1, \ldots, \iota_n\}$ and $\tau_i \vdash \tau'_j$ whenever $\iota'_j = \iota_i$ then

$$\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n \vdash \iota'_1 \mapsto \tau'_1 \wedge \cdots \wedge \iota'_m \mapsto \tau'_m,$$

where the ι'_j are distinct and the ι_i are distinct; the interpretation is

$$[\![\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n \vdash \iota'_1 \mapsto \tau'_1 \wedge \cdots \wedge \iota'_m \mapsto \tau'_m]\!]\langle v_1, \ldots, v_n\rangle = \langle v'_1, \ldots, v'_m\rangle$$

where $v'_j = [\![\tau_i \vdash \tau'_j]\!] v_i$ when $\iota'_j = \iota_i$.

Note that, because these coercions need not be injective, it is not possible to use a *generic* equality operator on product values; the interpretation of the equality would be ambiguous.

It is now possible to declare product-valued variables by using quantifiers of the form **new**$[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n]$. Values which are assignable to such variables can be created by using the following notation to append components to the value of a product-valued phrase:

product-value introduction:

$$\frac{V : \mathbf{val}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \rho_n] \quad V_j : \mathbf{val}[\tau'_j], \quad \text{for } 1 \leqslant j \leqslant m}{\langle V, \iota'_1 \mapsto V_1, \ldots, \iota'_m \mapsto V_m\rangle : \mathbf{val}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n \wedge \iota'_1 \mapsto \tau'_1 \wedge \cdots \wedge \iota'_m \mapsto \tau'_m]}$$

provided the ι'_j are distinct and $\iota'_j \neq \iota_i$ for $1 \leqslant j \leqslant m$, $1 \leqslant i \leqslant n$, and similarly for *expressions*; these rules should also be considered applicable when $V : \mathbf{val}[1]$, or $V : \mathbf{exp}[1]$, respectively. The interpretation is as follows:

$$[\![\langle V, \iota'_1 \mapsto V_1, \ldots, \iota'_m \mapsto V_m\rangle]\!] u$$

$$= \begin{cases} \bot, & \text{if } [\![V]\!]u = \bot \text{ or } [\![V_j]\!]u = \bot \text{ for any } 1 \leqslant j \leqslant m, \\ \langle \ldots, v_i, \ldots, v_j, \ldots\rangle, & \text{if } [\![V]\!]u = \langle \ldots, v_i, \ldots\rangle \text{ and} \\ & \quad [\![V_j]\!]u = v_j \neq \bot \text{ for } 1 \leqslant j \leqslant m, \end{cases}$$

and similarly for expressions. Notice that evaluation of a construction of this form forces the evaluation of *all* of its subphrases, whereas the tag-introduction notation combines *meanings*, such as (unevaluated) expressions. We allow

$$\langle \iota_1 \mapsto V_1, \ldots, \iota_n \mapsto V_n\rangle$$

as an abbreviation for

$$\langle \mathbf{nil}, \iota_1 \mapsto V_1, \ldots, \iota_n \mapsto V_n\rangle,$$

where **nil** : **val**[**1**] is a constant denoting the (only) value of data type **1**.

To allow selection of a component of a product-valued phrase we provide the following coercion:

$$\mathbf{val}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n] \vdash \iota_1 \mapsto \mathbf{val}[\tau_1] \wedge \cdots \wedge \iota_n \mapsto \mathbf{val}[\tau_n],$$

and similarly for expressions; then, conjunction-selection and tag elimination can

be used on the results. In fact, we can generalize this to *variables*, in order to allow "selective updating" of product-valued variables:

$$\mathbf{var}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n] \vdash \iota_1 \mapsto \mathbf{var}[\tau_1] \wedge \cdots \wedge \iota_n \mapsto \mathbf{var}[\tau_n].$$

The interpretations are as follows:

$$[\![\mathbf{val}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n] \vdash \iota_1 \mapsto \mathbf{val}[\tau_1] \wedge \cdots \wedge \iota_n \mapsto \mathbf{val}[\tau_n]]\!] v$$
$$= \langle \ldots, \iota_i \mapsto v_i', \ldots \rangle$$
$$\text{where } v_i' = \begin{cases} \bot, & \text{if } v = \bot, \\ v_i, & \text{if } v = \langle v_1, \ldots, v_n \rangle, \end{cases}$$

for $i = 1, 2, \ldots, n$, and similarly for expressions and

$$[\![\mathbf{var}[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n] \vdash \iota_1 \mapsto \mathbf{var}[\tau_1] \wedge \cdots \wedge \iota_n \mapsto \mathbf{var}[\tau_n]]\!] \langle a, e \rangle$$

$$= \langle \ldots, \iota_i \mapsto \langle a_i, e_i \rangle, \ldots \rangle,$$
$$\text{where } e_i(s) = \begin{cases} \bot, & \text{if } e(s) = \bot, \\ v_i, & \text{if } e(s) = \langle v_1, \ldots, v_n \rangle, \end{cases}$$
$$\text{and } a_i(e') = a(e''),$$
$$\text{where } e''(s) = \begin{cases} \bot, & \text{if } e(s) = \bot \text{ or } e'(s) = \bot, \\ \langle \ldots, v_{i-1}, e'(s), v_{i+1}, \ldots \rangle, & \text{if } e(s) = \langle v_1, \ldots, v_n \rangle \\ & \quad \text{and } e'(s) \neq \bot, \end{cases}$$

for $i = 1, 2, \ldots, n$. Then, a field selection of the form $V.\iota$ can be used as an acceptor when V is a product-valued variable, so that $V.\iota = E$ is equivalent to $V := \langle V, \iota \mapsto E \rangle$. The expression component as well as the acceptor component of the variable is needed to define the component acceptors.

7.2. Data-type syms

We allow for *sums* of data types as follows

$$\tau ::= \cdots \mid \iota \mathbin{@} \tau \mid \tau_1 \vee \tau_2, \quad \text{when } \tau_1 \bigtriangledown \tau_2,$$

where $\bigtriangledown$ (data-type disjoinability) is the smallest reflexive and symmetric binary relation on data-type expressions that satisfies

$$\text{if } \iota = \iota' \text{ implies } \tau \bigtriangledown \tau' \text{ then } \iota \mathbin{@} \tau \bigtriangledown \iota' \mathbin{@} \tau',$$
$$\text{if } \tau \bigtriangledown \tau_i \text{ for all } 1 \leq i \leq n \text{ then } \tau \bigtriangledown (\tau_1 \vee \cdots \vee \tau_n).$$

The appropriate coercions are

$$\text{if } \tau \vdash \tau' \text{ then } \iota \mathbin{@} \tau \vdash \iota \mathbin{@} \tau',$$
$$\iota \mathbin{@} \tau \vee \iota \mathbin{@} \tau' \vdash \iota \mathbin{@} (\tau \vee \tau'),$$
$$\tau_i \vdash \tau_1 \vee \tau_2 \quad \text{for } i = 1, 2,$$
$$\text{if } \tau_1 \vdash \tau, \tau_2 \vdash \tau \text{ then } \tau_1 \vee \tau_2 \vdash \tau.$$

There are no difficulties with least elements, and so the interpretation of data-type sums can be the obvious set-theoretic disjoint union. As in [9, 46], nontrivial coercions of data-type sums have the form

$$\text{if } \{\iota_1, \ldots, \iota_n\} \subseteq \{\iota'_1, \ldots, \iota'_m\} \text{ and } \tau_i \vdash \tau'_j \text{ whenever } \iota_i = \iota'_j \text{ then}$$

$$\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n \vdash \iota'_1 @ \tau'_1 \vee \cdots \vee \iota'_m @ \tau'_m$$

for ι_i distinct and ι'_j distinct; the interpretation is

$$[\![\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n \vdash \iota'_1 @ \tau'_1 \vee \cdots \vee \iota'_m @ \tau'_m]\!]\langle i, v\rangle$$

$$= \langle j, [\![\tau_i \vdash \tau'_j]\!] v\rangle, \quad \text{when } \iota_i = \iota'_j.$$

It is now possible to declare sum-valued variables by using quantifiers of the form $\mathbf{new}[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]$. Values assignable to such variables can be created by extending the $\langle \cdot \rangle$ notation as follows:

sum-value introduction:

$$\frac{V : \mathbf{val}[\tau]}{\langle \iota @ V\rangle : \mathbf{val}[\iota @ \tau]}$$

with semantics

$$[\![\langle \iota @ V\rangle]\!] u = \begin{cases} \bot, & \text{if } v = \bot, \\ \langle 1, v\rangle, & \text{otherwise} \end{cases}$$

where $v = [\![V]\!] u$,

and similarly for $\iota @ E$ when E is an expression. We can allow for discrimination among the possible variants for the value of a sum-valued value phrase by providing a coercion

$$\mathbf{val}[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n] \vdash \iota_1 @ \mathbf{val}[\tau_1] \vee \cdots \vee \iota_n @ \mathbf{val}[\tau_n]$$

whose interpretation is virtually an identity; however, a similar coercion for sum-valued *expressions* is not possible. Instead, we extend the applicability of the **casetag** construction as follows:

sum-valued expression elimination:

$$\frac{E : \mathbf{exp}[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]}{\mathbf{casetag}[\theta] E : (\iota_1 \mapsto (\mathbf{val}[\tau_1] \to \theta) \wedge \cdots \wedge \iota_n \mapsto (\mathbf{val}[\tau_n] \to \theta)) \to \theta}.$$

For $\theta' = \mathbf{exp}[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]$, the interpretation is as follows:

$$[\![\mathbf{casetag}[\theta] E]\!] u = \mathit{select}_{\theta' \theta}([\![E]\!] u)$$

where

$$select_{\theta'\theta}:[\![\theta']\!]\to[\![\iota_1\mapsto(\mathbf{val}[\tau_1]\to\theta)\wedge\cdots\wedge\iota_n\mapsto(\mathbf{var}[\tau_n]\to\theta)]\!]\to[\![\theta]\!]$$

is a family of functions defined by induction on θ, much like $cond_\theta$; for example,

$$select_{\theta'\mathbf{comm}}(e)\langle f_1,\ldots,f_n\rangle(s)=\begin{cases}\text{undefined}, & \text{if } e(s)=\perp,\\ f_i(v)(s), & \text{if } e(s)=\langle i,v\rangle,\end{cases}$$

and

$$select_{\theta'(\theta\to\theta'')}(e)\langle f_1,\ldots,f_n\rangle(a\in[\![\theta]\!])=select_{\theta'\theta''}(e)\langle f'_1,\ldots,f'_n\rangle$$
$$\text{where } f'_i(v\in[\![\theta_i]\!])=f_i(v)(a)\quad\text{for } 1\leqslant i\leqslant n,$$

and similarly for other types. Notice the similarity between the treatments of **if** and **casetag**: for "selectors" that are state-independent (e.g., of type **val[bool]** or a sum of phrase types), the "branches" can be of arbitrary phrase types; however, if the "selectors" are state-dependent (e.g., **exp[bool]** or a sum-valued expression), the "branches" must also be state-dependent (i.e., reducible by induction to an expression type or **comm**). In fact, the behaviour of **if** is essentially *derivable* from that of **casetag** by making the definitions

$$\mathbf{bool}\equiv \mathit{true}\,@\,\mathbf{1}\vee \mathit{false}\,@\,\mathbf{1}$$

$$\mathbf{true}\equiv\langle \mathit{true}\,@\,\mathbf{nil}\rangle,$$

$$\mathbf{false}\equiv\langle \mathit{false}\,@\,\mathbf{nil}\rangle,$$

$$\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \equiv\#(\mathbf{casetag}[\theta]B)\ \mathit{nil}.\mathit{true}\mapsto P_1\,|\,\mathit{false}\mapsto P_2.$$

A form of selective updating for sum-valued variables is also possible:

$$\mathbf{acc}[\iota_1\,@\,\tau_1\vee\cdots\vee\iota_n\,@\,\tau_n]\vdash\iota_1\mapsto\mathbf{acc}[\tau_1]\wedge\iota_n\mapsto\mathbf{acc}[\tau_n];$$

notice the change from sum to product. The interpretation is

$$[\![\mathbf{acc}[\iota_1\,@\,\tau_1\vee\cdots\vee\iota_n\,@\,\tau_n]\vdash\iota_1\mapsto\mathbf{acc}[\tau_1]\wedge\cdots\wedge\iota_n\mapsto\mathbf{acc}[\tau_n]]\!]a=\langle f_1,\ldots,f_n\rangle$$
$$\text{where } f_i(\iota_i)(e)=a(e')$$
$$\text{where } e'(s)=\langle i,e(s)\rangle$$

for $1\leqslant i\leqslant n$. Then, if V is a sum-valued variable and ι_i is one of the tags, $V.\iota_i:=E$ is equivalent to $V:=\langle\iota_i\,@\,E\rangle$. The expression part of the variable is not involved.

A "bottom" data-type name **0** is conceivable, but would be useless: $[\![\mathbf{0}]\!]$ would be the empty set.

7.3. Data-type arrays

We can also have data-type *arrays*; however, we cannot use procedures here and so we introduce new data-type expressions as follows:

$$\tau::=\cdots|[\sigma]\tau,$$

where σ is, as before, a statically-evaluable set-valued expression. Then,

$$[\![[\sigma]\tau]\!]=[\![\sigma]\!]\to[\![\tau]\!]$$

The coercions for array data types are as follows:

$$\text{if } [\![\sigma]\!] = [\![\sigma']\!] \text{ and } \tau \vdash \tau' \text{ then } [\sigma]\tau \vdash [\sigma']\tau',$$

with interpretation

$$[\![[\sigma]\tau \vdash [\sigma']\tau']\!]a = a; [\![\tau \vdash \tau']\!].$$

To create values which are assignable to array-valued variables, we extend the $\langle\cdot\rangle$ notation as follows:

array-value introduction:

$$\frac{V : \mathbf{val}[[\sigma]\tau] \quad V_1 : \mathbf{val}[\tau] \cdots V_n : \mathbf{val}[\tau]}{\langle V|[\sigma_1]V_1|\cdots|[\sigma_n]V_n\rangle : \mathbf{val}[[\sigma, \sigma_1, \ldots, \sigma_n]\tau]}$$

provided $[\![\sigma_i]\!] \cap [\![\sigma_j]\!] = \emptyset$ when $i \neq j$. The interpretation is

$$[\![\langle V|[\sigma_1]V_1|\cdots|[\sigma_n]V_n\rangle]\!]u$$

$$= \begin{cases} \perp, & \text{if } [\![V]\!]u = \perp \text{ or } [\![V_i]\!]u = \perp \text{ for any } 1 \leqslant i \leqslant n; \\ a, & \text{otherwise,} \end{cases}$$

$$\text{where } a(j) = \begin{cases} [\![V_i]\!]u, & \text{if } j \in [\![\sigma_i]\!] \\ [\![V]\!]uj, & \text{if } j \notin [\![\sigma_i]\!] \text{ for } 1 \leqslant i \leqslant n, \end{cases}$$

and similarly if V and the V_i are expressions. We can allow

$$\langle [\sigma_1]V_1|\cdots|[\sigma_n]V_n\rangle$$

as an abbreviation for

$$\langle \mathbf{undef}|[\sigma_1]V_1|\cdots|[\sigma_n]V_n\rangle.$$

We allow for selection on arrays of values and selective updating of array-valued variables by the coercions

$$\mathbf{val}[[\sigma]\tau] \vdash \mathbf{val}[\mathbf{nat}] \to \mathbf{val}[\tau],$$

$$\mathbf{exp}[[\sigma]\tau] \vdash \mathbf{exp}[\mathbf{nat}] \to \mathbf{exp}[\tau],$$

$$\mathbf{var}[[\sigma]\tau] \vdash \mathbf{exp}[\mathbf{nat}] \to \mathbf{var}[\tau],$$

with semantics

$$[\![\mathbf{val}[[\sigma]\tau] \vdash \mathbf{val}[\mathbf{nat}] \to \mathbf{val}[\tau]]\!]vv' = \begin{cases} \perp, & \text{if } v = \perp \text{ or } v' \notin [\![\sigma]\!], \\ v(v'), & \text{otherwise,} \end{cases}$$

and similarly for array-valued expressions, and

$$[\![\mathbf{var}[[\sigma]\tau] \vdash \mathbf{exp}[\mathbf{nat}] \to \mathbf{var}[\tau]]\!]\langle a, e\rangle(n) = \langle a_\tau, e_\tau\rangle$$

$$\text{where } e_\tau(s) = \begin{cases} \perp, & \text{if } e(s) = \perp \text{ or } n(s) \notin [\![\sigma]\!], \\ e(s)(n(s)), & \text{otherwise,} \end{cases}$$

$$\text{and } a_\tau(e') = a(e''),$$

$$\text{where } e''(s) = \begin{cases} \perp, & \text{if } e(s) = \perp \text{ or} \\ & \quad n(s) = \perp \text{ or } e'(s) = \perp, \\ (e(s) \mid n(s) \mapsto e'(s)), & \text{otherwise.} \end{cases}$$

Then, if A is an array-valued variable and ν is a numeral, $A(\nu) := E$ is equivalent to $A := \langle A \mid [\nu]E \rangle$.

7.4. Axioms

We now present some axioms valid for the data-type structuring facilities we have described. For V: **val**$[\tau]$ or V: **exp**$[\tau]$,

$$\langle \iota \mapsto V \rangle . \iota \equiv V,$$

$$\#(\textbf{casetag}[\textbf{val}[\tau]]\langle \iota_i @ V \rangle)\iota . \cdots \mid \iota_i \mapsto \iota \mid \cdots \equiv V,$$

$$\langle [\nu]V \rangle(\nu) \equiv V.$$

For ι_i distinct and V: **val**$[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n]$ or V: **exp**$[\iota_1 \mapsto \tau_1 \wedge \cdots \wedge \iota_n \mapsto \tau_n]$,

$$\langle \iota_1 \mapsto V.\iota_1, \ldots, \iota_n \mapsto V.\iota_n \rangle \equiv V.$$

For ι_i distinct and V: **val**$[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]$ or V: **exp**$[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]$,

$$\begin{aligned} &\#(\textbf{casetag}[\textbf{val}[\iota_1 @ \tau_1 \vee \cdots \vee \iota_n @ \tau_n]]V)\iota . \\ &\quad \iota_1 \mapsto \langle \iota_1 @ \iota \rangle \\ &\quad \mid \cdots \\ &\quad \mid \iota_n \mapsto \langle \iota_n @ \iota \rangle \\ &\quad \equiv V. \end{aligned}$$

For ν_i distinct and V: **val**$[[\nu_1, \ldots, \nu_n]\tau]$ or V: **exp**$[[\nu_1, \ldots, \nu_n]\tau]$,

$$\langle [\nu_1](V\nu_1) \mid \cdots \mid [\nu_n](V\nu_n) \rangle \equiv V.$$

For V: **var**$[\cdots \wedge \iota \mapsto \tau \wedge \cdots]$ and E: **val**$[\tau]$ or E: **exp**$[\tau]$,

$$V.\iota := E \equiv V := \langle V, \iota \mapsto E \rangle.$$

For A: **acc**$[\cdots \vee \iota @ \tau \vee \cdots]$ and E: **val**$[\tau]$ or E: **exp**$[\tau]$,

$$A.\iota := E \equiv A := \langle \iota @ E \rangle.$$

For V: **var**$[[\ldots, \nu, \ldots]\tau]$ and E: **val**$[\tau]$ or E: **exp**$[\tau]$,

$$V(\nu) := E \equiv V := \langle V \mid [\nu]E \rangle.$$

8. Discussion

Products, sums, and arrays have been discussed in many linguistic frameworks; for example, Carelli [9] uses the coercions on products, sums, and procedures

originally proposed in [44, 46] to model inheritance properties of object-oriented languages in an ML-like framework, and our language will have similar inheritance properties. However, Cardelli does not consider imperative aspects, or conventional coercions, such as de-referencing, whereas both applicative and imperative aspects have been treated here. Furthermore, as we argued in the introductory section, the essential characteristics of both "pure" styles of language are preserved in the ALGOL 60-like framework. This advantage comes at the cost of some complexity in requiring two levels of types, and in the treatment of conditionals. By comparison, languages in other frameworks may be superficially simpler; but, it should be much easier to reason about programs in an ALGOL-like language because both the laws of the lambda calculus and Hoare-like axioms are valid.

Of course, many issues must still be addressed before it can reasonably be claimed that we have a language that adequately supports both imperative and applicative programming. These include: recursive definitions of types, type abstraction, polymorphism, jumps, and references. For example, it should be possible for a programmer to define type-parameterized data types such as

$$ListOf[\tau] = null\,@\,\mathbf{1} \vee cons\,@\,(car \mapsto \tau \wedge cdr \mapsto ListOf[\tau]),$$

phrase types such as

$$LazyListOf[\theta] = null\,@\,\mathbf{val}[1] \vee pair\,@\,(first \mapsto \theta \wedge rest \mapsto LazyListOf[\theta]),$$

abstract types such as

$$\begin{aligned} StackOf[\tau]\ \mathbf{with}\ & clear \mapsto (StackOf[\tau] \to \mathbf{comm}) \\ & \wedge\ empty \mapsto (StackOf[\tau] \to \mathbf{exp}[\mathbf{bool}]) \\ & \wedge\ push \mapsto (\mathbf{exp}[\tau] \times StackOf[\tau] \to \mathbf{comm}), \\ & \wedge\ pop \mapsto (\mathbf{acc}[\tau] \times StackOf[\tau] \to \mathbf{comm}), \end{aligned}$$

and procedures such as

$$maplist[\tau_1, \tau_2]\colon (\mathbf{exp}[\tau_1] \to \mathbf{exp}[\tau_2]) \times \mathbf{exp}[ListOf[\tau_1]] \to \mathbf{exp}[ListOf[\tau_2]].$$

There has, of course, been considerable research on these issues [27, 30, 31, 46, 48, 49, 52, 60]. It remains to be seen how well these ideas can be integrated into the framework developed so far.

It would also be desirable to simplify and generalize the treatment of sum types, value phrases and conditionals. The value of a phrase of type $\mathbf{exp}[\tau]$ depends, potentially, on *all* of the storage state, whereas the value of a phrase of type $\mathbf{val}[\tau]$ can depend on *none* of the state. The obvious generalization is to refine the type system to allow the *degree* of state-dependence of a phrase to be expressed, with purely-applicative value phrases and totally-imperative expression phrases being the extreme points of the spectrum for expression-like phrases. If this could be extended to *all* phrase types, it might be possible to obtain simple and uniform semantics for conditional constructions such as **if** and **casetag** by constraining the

"selector" part to be no more state-dependent than the "branches". This proposal seems to require a generalization of the syntactic interference control described in [43].

Finally, the question of type inference for ALGOL-like languages should be studied. Although inference of virtually *all* type information, as in ML, does not seem feasible, it should be possible to find ways of reducing the amount of explicit type information necessary in some contexts. Some preliminary work along these lines for the polymorphic lambda calculus has been reported in [28].

Acknowledgement

I am grateful to John Reynolds for giving me access to notes on his unpublished work on conjunctive types, and for comments on an earlier presentation, and to Ellen Atack for many discussions on conjunctive types.

References

[1] P.W. Abrahams et al., The LISP 2 programming language and system, in: *Proceedings Fall Joint Computer Conference* (1966) 661-676.

[2] S. Abramsky and C.L. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, England, 1987).

[3] E. Atack, Conjunctive types in Algol-like languages, M.Sc. Thesis, Queen's University, Department of Computing and Information Science, Kingston, Ont. (1990).

[4] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (1978) 613-641.

[5] D.W. Barron et al., The main features of CPL, *Computer J.* **6** (1963) 134-143.

[6] F.L. Bauer et al., *The Munich Project CIP, 1: The Wide Spectrum Language CIP-L*, Lecture Notes in Computer Science **183** (Springer, Berlin, 1985).

[7] R. Bird and P. Wadler, *Introduction to Functional Programming* (Prentice-Hall, London, 1988).

[8] W.H. Burge, *Recursive Programming Techniques* (Addison-Wesley, Reading, MA, 1975).

[9] L. Cardelli, A semantics of multiple inheritance, *Inform. Comput.* **76** (1988) 138-164.

[10] M. Coppo and M. Dezani, A new type assignment for λ-terms, *Archiv. Math. Logik* **19** (1978) 139-156.

[11] L. Damas and R. Milner, 'Principle type-schemes for functional programs, in: *Conference Record 9th ACM Symposium on Principles of Programming Languages* (ACM, New York, 1982) 207-212.

[12] E.W. Dijkstra, Notes on structured programming, in: O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, eds., *Structured Programming* (Academic Press, London, 1972) 1-82.

[13] M. Felleisen, The calculi of lambda-v-CS-conversion: A syntactic theory of control and state in imperative higher-order programming languages, Ph.D. Dissertation, Computer Science Department, Indiana University, Bloomington, IN (1987).

[14] D.P. Friedman and D.S. Wise, CONS should not evaluate its arguments, in: S. Michaelson and R. Milner, eds., *Automata, Languages and Programming* (Edinburgh University Press, Edinburgh, 1976) 257-284.

[15] P. Henderson and J.H. Morris, A lazy evaluator, in: *Conference Record 3rd ACM Symposium on Principles of Programming Languages* (ACM, New York, 1976) 95-103.

[16] P. Henderson, *Functional Programming, Application and Implementation* (Prentice-Hall, London, 1980).

[17] H. Herrlich and G.E. Strecker, *Category Theory* (Heldermann, Berlin, 2nd ed., 1979).

[18] C.A.R. Hoare, Notes on data structuring, in: O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds., *Structured Programming* (Academic Press, London, 1972).
[19] C.A.R. Hoare, High-level languages, the way behind, in: D. Simpson, ed., *High Level Languages, The Way Ahead* (British Computer Society, NCC Publications, Manchester, 1973).
[20] C.A.R. Hoare, Hints on programming-language design, Tech. Rept. CS-74-403, Stanford University, Department of Computer Science, Stanford, CA (1974).
[21] C.A.R. Hoare, Recursive data structures, *Internat. J. Comput. Inform. Sci.* **4** (1975) 105-132.
[22] P. Hudak, Para-functional programming, *Computer* **19** (8) (1986) 60-71.
[23] M.B. Josephs, Functional programming with side effects, D. Phil. Thesis, Tech. Monograph PRG-55, Programming Research Group, Oxford University Computing Laboratory (1986).
[24] D.E. Knuth, The remaining troublespots in ALGOL 60, *Comm. ACM* **10** (10) (1967) 661-617.
[25] P.J. Landin, The next 700 programming languages, *Comm. ACM* **9** (3) (1966) 157-166.
[26] M. Lucassen, Types and effects: Towards the integration of functional and imperative programming, Ph.D. Thesis, Tech. Rept. MIT/LCS/TR-408, MIT Laboratory for Computer Science, Cambridge, MA (1987).
[27] N.J. McCracken, An investigation of a programming language with a polymorphic type structure, Ph.D. Dissertation, Syracuse University, Syracuse, NY (1979).
[28] N.J. McCracken, The typechecking of programs with implicit type structure, in: G. Kahn, D.B. MacQueen and G. Plotkin, eds., *Semantics of Data Types*, Lecture Notes in Computer Science **173**, (Springer, Berlin, 1984) 301-315.
[29] R. Milner, A proposal for standard ML, in: *Proceedings Symposium on LISP and Functional Programming* (ACM, New York, (1984) 184-197.
[30] J.C. Mitchell and R. Harper, The essence of ML, in: *Conference Record 15th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1988) 28-46.
[31] J.C. Mitchell and G.D. Plotkin, Abstract types have existential types, *ACM Trans. Programming Languages Syst.* **10** (3) (1988) 470-502.
[32] J.H. Morris, Real programming in functional languages, in: D.A. Turner, ed., *Functional Programming and Its Applications* (Cambridge University Press, Cambridge, 1982).
[33] P. Naur, ed., Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **6** (1) (1963) 1-20.
[34] F.J. Oles, A category-theoretic approach to the semantics of programming languages, Ph.D. Dissertation, Syracuse University, Syracuse, NY (1982).
[35] F.J. Oles, Type algebras, functor categories and block structure, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, Cambridge, 1985) 543-573.
[36] F.J. Oles, Lambda calculi with implicit type conversions, Computer Science Research Rept. RC 13245, IBM Research Division, T.J. Watson Research Center, Yorktorn Heights, NY (1987).
[37] D. Park, Some semantics for data structures, in: D. Michie, ed., *Machine Intelligence* **3** (Edinburgh University Press, Edinburgh, 1968), 351-371.
[38] G.D. Plotkin, Types and partial functions, Lecture Notes, Computer Science Department, University of Edinburgh, Edinburgh (1985).
[39] C.G. Ponder, P.C. McGeer and A.P.-C. Ng, Are applicative languages inefficient?, *SIGPLAN Notices* **23** (6) (1988) 135-139.
[40] G.J. Popek et al., Notes on the design of Euclid, *SIGPLAN Notices* **12** (3) (1977) 11-19.
[41] D. Prawitz, *Natural Deduction: A Proof-Theoretical Study* (Almquist and Wiksell, Stockholm, 1965).
[42] J.C. Reynolds, GEDANKEN: A simple typeless language based on the principle of completeness and the reference concept, *Comm. ACM* **13** (1970) 308-319.
[43] J.C. Reynolds, Syntactic control of interference, in: *Conference Record 5th ACM Symposium on Principles of Programming Languages* (ACM, New York, 1978) 39-46.
[44] J.C. Reynolds, Using category theory to design implicit conversions and generic operators, in: N.D. Jones, ed. *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science **94** (Springer, Berlin, 1980) 211-258.
[45] J.C. Reynolds, *The Craft of Programming* (Prentice-Hall, London, 1981).
[46] J.C. Reynolds, The essence of Algol, in: J.W. de Bakker and J.C. van Vliet, eds., *Algorithmic Languages* (North-Holland, Amsterdam, 1981) 345-372.
[47] J.C. Reynolds, Idealized Algol and its specification logic, in: D. Néel, ed., *Tools and Notions for Program Construction* (Cambridge University Press, Cambridge, 1982) 121-161.

[48] J.C. Reynolds, Types, abstraction and parametric polymorphism, in: R.E.A. Mason, ed., *Information Processing 83* (North-Holland, Amsterdam, 1983) 513–523.
[49] J.C. Reynolds, Three approaches to type structure, in: *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science **185** (Springer, Berlin, 1985) 97–138.
[50] J.C. Reynolds, Preliminary design of the programming language Forsythe, Tech. Rept. CMU-CS-88-159, Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1988).
[51] M. Schönfinkel, Uber die Bausteine der mathematischen Logik, *Math. Ann.* **92** (1924) 305–316.
[52] M.B. Smyth and G.D. Plotkin, The category-theoretic solution of recursive domain equations, *SIAM J. Comput.* **11** (1982) 761–783.
[53] J.E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).
[54] C. Strachey, Fundamental concepts in programming languages, Lecture Notes, International Summer School in Computer Programming, Copenhagen (1967).
[55] C. Strachey, The varieties of programming language, in: *Proceedings International Computing Symposium* (1972) 222–233, also: Tech. Monograph PRG-10, Programming Research Group, University of Oxford (1973).
[56] R.D. Tennent, *Principles of Programming Languages* (Prentice-Hall, London, 1981).
[57] R.D. Tennent, Quantification in Algol-like languages, *Inform. Process. Lett.* **25** (1987) 133–137.
[58] R.D. Tennent, Semantical analysis of specification logic, *Inform. and Comput.* (to appear).
[59] R.D. Tennent, Denotational semantics, in: S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, eds., *Semantics, Handbook of Logic in Computer Science II* (Oxford University Press, Oxford, 1990).
[60] R.D. Tennent and K. Tobin, Continuations in possible-world semantics, *Theoret. Comput. Sci.* (to appear).
[61] M. Tofte, Operational semantics and polymorphic type inference, Ph.D. Thesis, Tech. Rept. CST-52-88, Department of Computer Science, University of Edinburgh, Edinburgh (1988).
[62] D. van Dalen, *Logic and structure* (Springer, Berlin, 2nd ed., 1983)
[63] A. van Wijngaarden et al., Revised report on the algorithmic language Algol 68, *Acta Inform.* **5** (1975) 1–236.
[64] W.W. Wadge and E.A. Ashcroft, *Lucid, the Dataflow Programming Language*, APIC Studies in Data Processing **22** (Academic Press, London, 1985).
[65] C.P. Wadsworth, Semantics and pragmatics of the lambda calculus, D. Phil. Dissertation, University of Oxford (1971).
[66] N. Wirth, On the design of programming languages, in: J.L. Rosenfeld, ed., *Proceedings IFIP Congress 74*, Stockholm (North-Holland, Amsterdam, 1974) 386–393.