Rationale for O'Haskell

Introduction

The construction of robust distributed and interactive software is still a challenging task, despite the recent popularity-increase for languages that take advanced programming concepts into the mainstream. Several problematic areas can be identified: most languages require the **reactivity** of a system to be manually upheld by careful avoidance of blocking operations; mathematical **values** often need to be encoded in terms of stateful **objects** or vice versa; **concurrency** is particularly tricky in conjunction with encapsulated software components; and static type safety is often compromised because of the lack of simultaneous support for both **subtyping** and **polymorphism**.

This document presents a rationale for a programming language, *O'Haskell*, that has been consciously designed with these considerations in mind. O'Haskell is defined by conservatively extending the purely functional language Haskell with the following features:

- A central structuring mechanism based on *reactive objects*, which unify the notions of objects and concurrent processes. Reactive objects are asynchronous, state-encapsulating servers whose purpose is to react to input messages; they cannot actively block execution or selectively filter their sources of input.
- A monadic layer of object-based computational effects, which clearly separates stateful objects from stateless values. Apart from higher-order functions and recursive data structures, the latter notion also includes first-class commands, object templates, and methods.
- A safe, polymorphic type system with declared record and datatype subtyping, supported by a powerful partial type inference algorithm.

It is claimed that these features make O'Haskell especially well-adapted for the task of modern software construction.

Reactive objects

Object modeling

The *object model* [1] offers a remarkably good strategy for decomposing a complex system state into a web of more manageable units: the state-encapsulating, identity-carrying entities called objects. Objects are abstractions of autonomous components in the real world, characterized by the shape of their internal state and the *methods* that define how they react when exposed to external stimuli. The object model thus inherently recognizes a defining aspect of interactive computing: systems do not terminate, they just maintain their state awaiting further interactions [2]. Not surprisingly, object modeling has become the strategy of choice for numerous programming tasks, not least those that must interface with the concrete state of the external world. For this reason, objects make a particularly good complement to the abstract, stateless ideal of functional programming.

The informal object model is naturally concurrent, due to the simple fact that real world objects communicate and ``execute their methods'' in parallel. On this informal plane, the intuition behind an object is also entirely reactive: its normal, passive state is only temporarily interrupted by active phases of method execution in response to external requests. Concurrent object-oriented languages, however, generally introduce a third form of state for an object that contradicts this intuition: the active, but *indefinitely indisposed* state that results when an object executes a call to a disabled method.

The view of indefinite blocking as a transparent operational property dates back to the era of batch-oriented computing, when interactivity was a term yet unheard of, and buffering operating systems had just become widely employed to relieve the programmer from the intricacies of synchronization with card-readers and line-printers. Procedure-oriented languages have followed this course ever since, by maintaining the abstraction that a program environment is essentially just a subroutine that can be expected to return a result whenever the program so demands. Selective method filtering is the object-oriented continuation of this tradition, now interpreted as ``programmers are more interested in hiding the intricacies of method-call synchronization, than preserving the intuitive responsiveness of the object model''.

Some tasks, like the standard bounded buffer, are arguably easier to implement using selective disabling and queuing of method invocations. But this help is deceptive. For many clients that are themselves servers, the risk of becoming blocked on a request may be just as bad as being forced into using *polling* for synchronization, especially in a distributed setting that must take partial failures into account. Moreover, what to the naive object implementor might look like a protocol for imposing an order on method invocations, is really a mechanism for *reordering* the invocation-sequences that have actually occurred. In other words, servers for complicated interaction protocols become disproportionately easy to write using selective filtering, at the price of making the clients extremely sensitive to temporal restrictions that may be hard to express, and virtually impossible to enforce.

Existing concurrent object-oriented languages tend to address these complications with an even higher dose of advanced language features, including path expressions [3], internal concurrency with threads and locks [4, 5], delegated method calls [6], future and express mode messages [7], secretary objects [8], explicit queue-management [6, 9], and reification/reflection [10]. O'Haskell, the language we put forward in this document, should be seen as a minimalistic reaction to this trend.

Preserving reactivity

The fundamental notion of O'Haskell is the *reactive object*, which unifies the object and process concepts into a single, autonomous identity-carrying entity. A reactive object is a passive, state-encapsulating server that also constitutes an implicit critical section, that is, at most one of its methods can be active at any time. The characteristic property of a reactive object is that its methods *cannot* be selectively disabled, nor can a method choose to wait for anything else than the termination of another method invocation. The combined result of these restrictions is that in the absence of deadlocks and infinite loops, objects are indeed just *transiently* active, and can be guaranteed to react to method invocations within any prescribed time quantum, just given a sufficiently fast processor. Liveness is thus an intrinsic property of a reactive object, and for that reason, we think O'Haskell represents a

quite faithful implementation of the intuitive object model.

Concurrent execution is introduced in O'Haskell by invoking *asynchronous* methods, which let the caller continue immediately instead of waiting for a result. A blocking method call must generally be simulated by supplying a *callback* method to the receiver, which is the software equivalent of enclosing a stamped, self-addressed envelope within a postal request. Two additional features of O'Haskell contribute to making this convention practically feasible. Firstly, methods have the status of first-class values, which means that they can be passed as parameters, returned as results, and stored in data structures. Secondly, the specification of callbacks, their arguments, and other prescribed details of an interaction interface, can be concisely expressed using the statically safe type system of the language.

There is, however, an important subcase of the general send/reply pattern that does not require the full flexibility of a callback. If the invoked method is able to produce a reply in direct response to its current invocation, then the invoking object may safely be put on hold for this reply, since there is nothing but a fatal error condition that may keep it from being delivered. For such cases, O'Haskell offers a *synchronous* form of method definitions, thus making it possible to syntactically identify the value-returning methods of foreign objects that *truly* can be called just as if they were subroutines.

With these contrasting forms of communication in mind, we may draw an analogy to the cost-effective, stress-reducing coordination behaviour personified by a top-rated butler(!). Good butlers ask their masters to standby only if a requested task can be carried out immediately and under full personal control (or, if necessary, using only the assistance of equally trusted servants). In all other cases, a good butler should respond with something like an unobtrusive ``of course'' to assure the master that the task will be carried out without the need for further supervision, and in applicable cases, dutifully note where and how the master wants any results to be delivered. Only an extraordinarily bad butler would detain his master with a ``one moment, please'', and then naively spend the rest of the day waiting for an event (e.g. a phone call) that may never come.

Reactive objects in O'Haskell allow only ``good butlers'' to be defined. We believe that this property is vital to the construction of software that is robust even in the presence of such complicating factors as inexperienced users, constantly changing system requirements, and/or unreliable, long-latency networks like the Internet. As an additional bonus, reactive objects are simple enough to allow both a straightforward formal definition and an efficient implementation. Still, the model of reactive objects is sufficiently abstract to make concurrency an intuitive and uncomplicated ingredient in mainstream programming, something which cannot be said about many languages which, like Java, offer concurrency only as an optional, low-level feature.

Reactivity in interfaces

One consequence of the restricted synchronization mechanisms of O'Haskell is that changing the implementation of a service, from a strictly local computation to a request involving network access, cannot be done without modifying the interface to that service. For example, if a service previously was implemented as as synchronous request, making the result dependent on packets arriving over a network can only be achieved by turning the service into an asynchronous method which takes a parameterized callback method as an argument. We argue that this is as it should be, since the semantics of the service also changes, from that of a simple subroutine call, to the uncertainty inherent in a synchronizing operation.

Such a change in semantics can have, and should have, consequences to the whole structure of the user of a service. This fact is well illustrated by programs that utilize Sun's distributed file system NFS. For pragmatic reasons, Sun decided not to change the file system interface when going from a local implementation to a distributed design, sensitive to partial failures. Since file system clients accordingly did not need to change either, most programs that access NFS remain totally unprepared for the fact that a remote file server may go down, which shows up to the user as a *total* loss of

responsiveness in those situations [11]. Another example on the same topic is given by common web-browsers. These programs are generally carefully designed to allow continuous interaction even if a remote web server is not responding. However, most browser implementations fail to notice the fact that name-lookup of web addresses usually takes the shape of a remote service as well, and hence their user-interfaces freeze completely in the event of an inaccessible name server. Such a mistake would not be possible in a language where indefinite blocking cannot be masqueraded as a simple method call.

In fact, revealing the potential for indefinite blocking by requiring a callback is just a continuation of the argument that operational properties should be captured by types. Monadic programmers are already used to the idea that changing a pure function by adding side-effects should not be possible without simultaneously changing the type of the function as well. Reactive objects are simply an application of this idea to the *temporal* aspects of computing.

Having that said, we would also like to point out that even established object-oriented practice is characterized by a desire to keep the message-passing metaphor free from blocking complications as far as possible, e.g. by attempting to concentrate all uses of potentially blocking operations to the head of a so-called *event-loop*. Programming styles developed with these considerations in mind are thus immediately expressible in O'Haskell, even without the introduction of callbacks. In fact, many common services do not even return a reply (e.g. methods that implement event-signaling or simple data pipelining), so in these cases simple asynchronous communication would suffice. The consequence of preserving reactivity primarily manifests itself in the encoding of classical synchronization primitives like semaphores and channels. Programming with explicit synchronization operations seems to require a major rethink in O'Haskell, a topic which is illustrated on the O'Haskell programming examples page.

Preserving message ordering

A natural consequence of not allowing selective message reordering is to prescribe that methods should be executed in the order they are invoked, whenever that order can be uniquely determined. This is also exactly what O'Haskell does; i.e. an object can count on that two consecutive messages to the same peer will be handled in the order they are sent. However, common practice in concurrent languages is to leave this issue unspecified, in order to facilitate distributed implementations across an unreliable network [12, 13]. We take on our opposing standpoint on basis of the following arguments.

- Many simple programs would be unduly complicated if message ordering was not preserved (just imagine communicating with a storage-cell under such premises).
- Processes on an unreliable network can be conveniently accessed via a library of local *proxy* objects. These proxies can present a reliable or an unreliable connection, at the free choice of the implementor. Implementing a proxy is greatly simplified, however, if communication with the local clients is order preserving.
- The only operation which cannot be faithfully simulated by a proxy is the synchronous request. Thus, a non-local object will need to have a more limited interface than a corresponding local one. This is a necessary restriction if we consider liveness to be important, and indefinitely blocking input to be harmful. On the other hand, if the network in question is considered so reliable that the lack of synchronous requests is just perceived as an annoying constraint, then the network can equally well be made a part of the language implementation, since it guarantees the language semantics.

Values vs. Objects

Values in object-oriented languages

Most object-oriented languages provide very little support for data structuring tools besides objects. The reason for this sparseness is of course conceptual economy (the *everything-is-an-object* metaphor), but it also has as a consequence that every data entity must have the properties inherent in the object model, i.e. a *state*, an *age*, a *location*, an *identity*. For some kinds of data this point of view is entirely inappropriate - the value 7, for example, was not ``created'' at some point in time, neither can we make it cease to exist, nor count how many 7's there are. Moreover, adding 2 to 5 does not update the meaning of 5 or 2 in any way, nor does it produce a ``new'' 7. What mathematics teaches us is that numbers are abstract, timeless entities, and that computing with numbers (for example by adding them) is just a mechanical way of simplifying an expression to the canonical form that denotes its value.

Accordingly, all but the most spartan object-oriented languages give basic types such as numbers some special treatment that more closely matches our intuition. But there is a wealth of mathematically inspired structures that suffer from the same conceptual mismatch - these include pairs, lists, records, functions, and algebraic datatypes. Encoding such structures in terms of stateful objects is not only tiresome and error-prone, it may also prohibit efficient and safe sharing of data.

As an example, consider an encoding of vectors in space in terms of objects. In such an encoding, arbitrary vector values cannot simply be written as canonical expressions when needed, because values are represented by object references that must be created at some point in time by an instantiation command. Moreover, ordinary equality on these references actually makes it possible to discriminate values on basis of their location, i.e. to distinguish between *this* origin and *that* origin, etc. Hence the programmer must make sure that special purpose code gets called whenever a mathematical notion of vector equality is desired. And not the least, the programmer must also resist the temptation to implement a vector operation by destructively updating the state of the called object. If not, the meaning of vector values will indeed vary with the operations being performed, which is just as illogical for vectors as it is for numbers. Yet in an imperative language, destructive updating may often be the only reasonable implementation alternative, especially in the encoding of slightly more complex structures such as dynamic lists. Sharing stateful values with unknown code is obviously unsafe from this point of view, all the more if the unknown code happens to be a concurrently executing process. So this leaves the programmer with the choice of either conservatively cloning local objects that represent invariant values before sending them off to an unknown context, or (which is more likely) circumscribing stateful messages with informal restrictions on what can be done with them, and then simply hope for the best.

Functional programming

In contrast to the problem of finding good representations for values in object-oriented languages, functional programming builds directly upon a recognition of the mathematical nature of values. The functional paradigm advocates declarative construction, analysis, and simplification of arbitrary complex value expressions as the primary means of expressing a computational task. Variables in functional languages always denote constants, and in purely functional languages like Haskell, simplification of an expression is guaranteed not to have any computational effect whatsoever, besides termination with a canonical result. Among other things, this simple semantics opens up many possibilities for liberal sharing of data, since no program activity can affect the meaning of a variable, and no variable can be introduced that affects the meaning of a given program.

This property also goes under the name of *referential transparency*, and is often considered to be the defining characteristic of a purely functional language. Referential transparency enables the use of equational reasoning as a program verification technique, which indeed is a very strong argument in favour of a declarative programming style. Additional benefits commonly attributed to functional programming include a succinct notation, an efficient form of declaration by pattern-matching, and the important ability to treat functions themselves as first-class data values [14].

Purely functional programs are however inherently weak in expressing interaction. Their calculator-like model of computation intuitively suggests that *input* is a single value that somehow happens to be available at program initiation, and that *output* can be limited to a single value returned at termination. Interacting objects can at best be modeled indirectly, but then only by means of encodings that reintroduce terms like state, change, and identity in the model - i.e. the very notions that advocates of functional programming deliberately have shun. Besides that, a fundamental problem with the functional model is that it cannot host the *non-determinism* of concurrently executing objects without a substantial repercussion on either its semantics or pragmatics [15, 16].

The functional and object-oriented paradigms achieve conceptual efficiency by means of purification in terms of *either* values or objects. Yet both views are indispensable in modern programming [17]. Values and value-oriented programming capture the fact that the computer indeed is a fast and flexible calculator, whereas objects and object-oriented programming focus on the storage capacity of computers, and the need for well-structured interaction with this storage. Value-oriented programming manifests computation by means of *expressions* that *denote a result*, while object-oriented programming uses *commands* that, when executed, will *cause an effect*. Values in this sense represent ideas, or abstractions in our minds, that have a name but no definition. Objects, on the other hand, represent our interpretation of concrete things in the world around us, as we understand them in terms of their constituent parts and their dynamic behaviour. The programming style for computing with values is thus inherently declarative, whereas the nature of object-oriented programming is intrinsically imperative. And just as declarative *thinking* and imperative *doing* are complementary aspects of our daily life, so are the declarative and imperative styles when it comes to modern computer programming. Fortunately, the discovery of *monads* has made it possible for a programming language to be truly declarative and truly imperative at the same time. We will capitalize on this achievement in our integration of value- and object-oriented programming in O'Haskell.

A monadic approach to objects

Monads [18, 19] are becoming widely recognized as the de facto standard for conservatively extending a declarative language with computational effects. The monadic approach is predominantly associated with Haskell and its imperative I/O system [20], but the technique is by no means limited to this language or to a particular evaluation strategy [21, 22], not even to languages that exclusively belong to the functional category [23].

Monads have been successfully utilized to incorporate a quite diverse set of classically imperative features in Haskell, including traditional I/O [20], graphical user interfaces [24, 25], first-class pointers [20], concurrent processes and synchronization variables [13], and exceptions [26]. The common denominator in all these proposals is the unifying property of a *stratified semantics*, that limits the effect of imperative commands to the top-level of a program, while leaving the purely declarative semantics of expression evaluation unaffected. This means that a monadic command under execution may very well request the evaluation of an expression, but an expression being evaluated can never trigger the execution of a command.

O'Haskell brings the world of object-based concurrency into the realm of monadic functional programming. Compared to other monad-based Haskell extensions, O'Haskell represents a higher level of abstraction, by its use of objects instead of pointers, and methods instead of peeks and pokes as the basic imperative building blocks. The high-level flavour of O'Haskell is also evident in its syntax for monadic programming, that improves on Haskell by also taking the need for unconvoluted use of assignable state variables into account.

The essence of the monadic approach is the insight that *doing* something and merely *thinking about doing* something are radically different activities. Translated into programming terms, this means that an imperative command is also considered to be a first-class declarative value, although a value of an abstract type (the monad) that just represents the hypothetical effect of a command, *should it ever be*

executed. Command values can thus be declaratively combined, named, parameterized, and calculated with just like any other value, since the evaluation of a command expression is kept entirely distinct from the actual *realization* of the effect it represents.

All commands in O'Haskell, including those that refer or assign to state variables, enjoy the status of first-class values. Furthermore, since any command can be individually named, the name of a method invocation may equally well be considered the name of the actual method it invokes, just as the name of an object creation command may be identified with the *class* of objects it instantiates. Hence we are able to promote even methods, as well as object templates, to full-blown value status in O'Haskell (O'Haskell actually uses the term *template* instead of class to avoid confusion with the established Haskell concept of *type classes*).

This last property makes parameterization over callback methods easy. In fact, higher-order parameterization as a general programming technique has the potential of facilitating very simple solutions to many problems that immediately seem to lead to quite contorted encodings in established object-oriented languages. Examples of this include parameterization of an object template over (i) a set of neighbouring objects, (ii) other templates from which private but unknown sub-objects may be created, or (iii) a single command denoting a yet undefined aspect of an object's behaviour. Other speculative examples are methods creating object instances from a supplied template parameter on behalf of external clients, methods taking a communication pattern as a command argument, or simply a method parameterized over a function value that expresses some unknown computation on the local state. This flexibility should be compared to the complicated exercise in inheritance and override that is required in an object-oriented language like Java to express even such a simple concept as a callback parameter.

The communication interface returned when an object is created is also a full-fledged value in O'Haskell, that may or may not reveal information about the identity of the object behind the interface. Most likely an object interface is defined as a record of method values, but it might also be a tuple of such records, or even a function that returns different interfaces depending on a supplied argument such as a password! Separating values from stateful objects opens up many interesting possibilities, as we hope our coding examples will demonstrate.

Subtyping and polymorphism

Type systems for functional and object-oriented languages have evolved along two quite orthogonal lines. Mainly due to the influential type system of Standard ML [27], functional type systems in general have acquired a certain flavour that can be characterized by a preference for the algebraic datatype (the *labeled sum*) as the principal data structure, and a widely employed use of *parametric polymorphism* as the primary means of achieving typing flexibility. This is matched in modern object-oriented type systems by an equally strong preference for data structures based on records (*labeled products*), and a tradition of obtaining flexible typings through the use of *subtyping* (again mostly due to a single seminal language: Simula [28]). Other, mostly one-dimensional distinctions can also be identified, for example concerning the questions of automatic type inference vs. explicit typing, or static type safety vs. optional run-time type inspection. Still, these minor observations do not really contradict the fact that functional and object-oriented type system are essentially characterized by features that are by no means mutually exclusive.

The predominance of a particular kind of type-forming operator in each school is not a coincidence, though [29]. Programming is essentially the task of manipulating data structures, and different parts of a program take the roles of either data-producers or data-consumers in this play. The functional approach to programming is to ask ``how is data constructed?''. This leads to a style of programming where the data *constructors* are considered primitive, and where data-consuming code accordingly is defined by pattern-matching over these constructors. The object-oriented programmer instead starts

out by asking ``what can we do with the data?'', which fosters the view that it is the data *selectors* that should be seen as foundational, and that the code of a data-producer consequently should be defined as an enumeration of implementations for these selectors. (Here already lies the key to the distinctive form of data abstraction that goes under the name object-orientation: the producer of a data object gets full control over what the result of consuming the object will be - the consumer is effectively hindered from making that decision on basis of how the object was constructed.)

The preference for a certain form of data structures also reflects a particular view of where extensibility is most needed. The functional style ensures that adding another data-consumer (a function) is a strictly local change, whereas adding a new data-producer requires extending a datatype with a new constructor, and thus a major overhaul of every pattern-matching function definition. Likewise, in object-oriented programming the addition of a new data-producer (a new class) is a cinch, while the introduction of another data-consumer means the extension of an record type with a new selector, a global undertaking with implications to potentially all existing classes.

To alleviate these drawbacks to some extent, functional and object-oriented traditions prescribe their own custom methods. Parametric polymorphism in functional languages allows functions to be defined uniformly for arbitrary sets of data constructors, including constructors not yet visualized. Object-oriented subtyping, on the other hand, facilitates incremental definition of record types, so that old data-consuming code still can be reused even on data objects defined by a larger set of selectors than what was originally envisaged. The latter openness to future refinements in design is actually the intuition behind the unmistakable *is-a*-relation that forms a central part of the biological metaphor permeating object-oriented classification [30]. However, as is well known, nothing really precludes the merits of subtyping from being applied to the definition of datatypes as well; or for that matter, polymorphism to be utilized in conjunction with records [31].

As a hybrid language, O'Haskell attempts to bring the design issues that accompany constructor- or selector-based programming to the fore, so that the programmer can choose the right kind of data structure for each task. Hence the type system of O'Haskell offers equal opportunities for working with labeled sums and labeled products. Polymorphism is available in both cases, and so is the possibility of defining types incrementally, (i.e. subtyping is supported for both parameterized records and parameterized datatypes). Furthermore, neither kind of data structure is tied to any particular choice between declarative and imperative programming, nor is the availability of type inference dependent on whether a datatype or a record type is used. However, type inference in conjunction with subtyping holds its special set of problems, which have been given a somewhat unconventional, yet very effective solution in O'Haskell.

Subtyping and type inference

The combination of subtyping with polymorphic type inference has been under intensive study for more than a decade [32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43]. From the outset, this line of research has focused on *complete* inference algorithms, and the related notion of principal types. This direction is not hard to justify considering the evident merits of the Hindley/Milner type system that underlies polymorphic languages like Haskell or Standard ML: program fragments can be typed independently of their context, and programmers may rest assured that any absent type information will be filled in with types that are at least as general, and at least as succinct, as any information the programmer would have come up with.

Still, although complete algorithms for polymorphic subtype inference exist, practical language implementations that take advantage of these are in short supply. The main reason seems to be that ``the algorithms are inefficient and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read'' [44]. Many attempts have been made at simplifying the output from these algorithms, but they have only partially succeeded, since the problem in its generality seems to be intractable, both in theory and practice [44, 41].

However, even if type simplification were not an issue, there is an inherent conflict between generality and succinctness in polymorphic subtyping that is not present in the original Hindley/Milner type system. While the principal type of (say) a Standard ML expression is also the syntactically shortest type, the existence of subtype constraints in polymorphic subtyping generally makes a principal type *longer* than its instances. In particular, the principal type for a given expression may be substantially more complex than the simplest type general enough to cover an *intended* set of instances! Thus, type annotations, which give the programmer direct control over the types of expressions, are likely to play a more active role in languages with polymorphic subtyping than they do in Haskell or Standard ML, irrespective of advances in simplification technology.

In the design of a type system for O'Haskell, we have taken a pragmatic standpoint and embraced type annotations as a fact of life. This has enabled us to focus on the much simpler problem of *partial* polymorphic subtype inference. As one of the main contributions of the language, we present an inference algorithm for our type system that always infers types without subtype constraints, if it succeeds. This is a particularly interesting compromise between implicit and explicit typing, since such types possess the desirable property of being syntactically shorter than their instances, even though they might not be most general. We might say that the algorithm favours readability over generality, leaving it to the programmer to put in type annotations where this strategy is not appropriate.

Our inference algorithm is based on an approximating constraint solver, that resorts to ordinary unification when two type variables are compared. An exact characterization of which programs the algorithm actually is able to accept is still an open problem, but we prove, as a lower bound, that it is complete with respect to the Hindley/Milner type system, as well as the basic type system of Java. The algorithm is furthermore both efficient and easy to implement, and as our programming examples will indicate, explicit type-annotations are rarely needed in practice.

Name inequivalence

An additional contribution of the O'Haskell type system is presumably the subtype relation itself, which is based on *name inequivalence*, in contrast to the structural ditto that dominate the standard literature [31, 32]. By structural we mean the common practice of defining special subtyping rules for functions, records, and variants, etc, assuming a given partial order over a set of nullary base types. Being a conservative extension to the type system of Haskell, our system instead assumes that type constants can be of any arity, and the partial order on base types is accordingly replaced by a relation on fully saturated applications of these constants. Our subtyping relation furthermore supports polymorphic instantiation of the basic subtype axioms, and a notion of *depth* subtyping that works inside type constant applications on the basis of inferred *variances* for the type parameters.

We believe that working with named and parameterized types, whose subtype relationships are defined by declaration, has the immediate benefit of giving the programmer full control over the type structure of a program. This can be a valuable tool in the design phase of a large system, and it also offers greater protection from certain logical errors. Furthermore, name inequivalence is more in line with both the way datatypes are treated in Haskell and other functional languages, and with the object-oriented notion of subtyping between named classes. And not the least, the ability to refer to types by name has a big impact on the readability of the output from our inference algorithm.

References

[1] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, 1990.

[2] Peter Wegner. Interactive software technology. In Allen B. Tucker, editor, *Handbook of*

Computer Science and Engineering. CRC Press, 1996.
[3] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems, Proc.*, volume 16 of *LNCS*, pages 89--102. Springer Verlag, April 1974.
[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, 1985.
[5] Gregory R. Andrews and Ronald A. Olsson. *The SR programming language*. The Benjamin/Cummings Publishing Company, 1993.
[6] O. M. Nierstrasz. Active objects in hybrid. *ACM SIGPLAN Notices*, 22(12):243--253, December 1987.
[7] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 55--89. MIT Press, Cambridge, MA, 1987.
[8] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices*, 22(12):406--415, December 1987.
[9] Anand Tripathi and Mehmet Aksit. Communication, scheduling and resource management in SINA. *Journal of Object-Oriented Programming*, 2(4):24--36, November 1988.
[10] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. *ACM SIGPLAN Notices*, 23(11):306--315, November 1988.
[11] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.
[12] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
[13] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295--308, St Petersburg, FL, January 1996. ACM Press.
[14] D. A. Turner. The Semantic Elegance of Applicative Languages. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.
[15] A. K. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University of Technology, G–teborg, Sweden, September 1998.
[16] F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, 31(3):243--247, June 1988.
[17] B. MacLennan. Values and Objects in Programming Languages. *ACM SIGPLAN Notices*, 17 (12):70--80, 1982.
[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55--92, 1991.
[19] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.
[20] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293--341, December 1995.
[21] Lennart Augustsson. Cayenne -- a language with dependent types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 239--250, September 1998.
[22] A. Filinski. Representing monads. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 446--457, New York, NY, USA, 1994. ACM Press.
[23] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.
[24] K. Claessen, E. Meijer, and T. Vullinghs. Implementing Graphical Paradigms in TkGofer. In *Proceedings of the 1997 International Conference on Functional Programming*, Amsterdam, 1997. ACM.
[25] Sigbjorn Finne and Simon Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastrict, Netherlands, September 1995.
[26] Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A

semantics for imprecise exceptions. In *Proc. Programming Languages Design and Implementation (PLDI'99)*, Atlanta, GA, 1999.
[27] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
[28] Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard. SIMULA 67. common base language. Technical Report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.
[29] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990, volume 489 of *Lecture Notes in Computer Science*, pages 151--178. Springer-Verlag, New York, N.Y., 1991.
[30] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.
[31] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.
[32] J. Mitchell. Coercion and type inference. In *ACM Principles of Programming Languages*, 1984.
[33] Y. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73, 1990.
[34] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Theory and Practice of Software Development*, Barcelona, Spain, March 1989. Springer Verlag.
[35] J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 36 (3):245--285, 1991.
[36] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Lisp and Functional Programming*, pages 193--204, San Francisco, CA, June 1992.
[37] G. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, (23):197--226, 1994.
[38] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993.
[39] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA '95*. ACM, 1995.
[40] Fritz Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, May 1996.
[41] J. Rehof. Minimal typings in atomic subtyping. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.
[42] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136--149, Amsterdam,Holland, June 1997.
[43] Dilip Sequeira. *Type Inference with Bounded Quantification*. PhD thesis, University of Edinburgh, 1998.
[44] M. Hoang and J. Mitchell. Lower Bounds on Type Inference With Subtypes. In *ACM Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.

[Back to the O'Haskell homepage]

Page maintained by Johan Nordlander. Last modified: January 26 2001