# Dynamic Typing as Staged Type Inference

Mark Shields*
University of Glasgow
mbs@dcs.glasgow.ac.uk

Tim Sheard†
Oregon Graduate Institute
sheard@cse.ogi.edu

Simon Peyton Jones‡
University of Glasgow
simonpj@dcs.glasgow.ac.uk

### Abstract

Dynamic typing extends statically typed languages with a universal datatype, simplifying programs which must manipulate other programs as data, such as distributed, persistent, interpretive and generic programs. Current approaches, however, limit the use of polymorphism in dynamic values, and can be syntactically awkward.

We introduce a new approach to dynamic typing, based on *staged computation*, which allows a single type-reconstruction algorithm to execute partly at compile time and partly at run-time. This approach seamlessly extends a single type system to accommodate types that are only known at run-time, while still supporting both type inference and polymorphism. The system is significantly more expressive than other approaches. Furthermore it can be implemented efficiently; most of the type inference is done at compile-time, leaving only some residual unification for run-time.

We demonstrate our approach by examples in a small polymorphic functional language, and present its type system, type reconstruction algorithm, and operational semantics. Our proposal could also be readily adapted to many other programming languages.

## 1 Introduction

Sometimes static typing can be too static. Type information may either be difficult to express, or be unknown until run-time. Consider replicating C's *sprintf* function in a functional language. We would like to write

$$\textit{sprintf} \text{ “\%n = \%b” } (1, \mathbf{true})$$

where “%n” and “%b” are placeholders for the elements of the argument tuple. Unfortunately giving *sprintf* a type such as

$$\textit{sprintf} : \mathbf{String} \to \tau \to \mathbf{String}$$

is problematic, as the type $\tau$ depends on the value of *sprintf*'s first argument.

Examples such as this are common in:

- Persistent programming, where values of any type may be stored and retrieved from stable storage.
- Distributed programming, where data and code are exchanged between remote programs.
- Interpretive programming, where object language terms of arbitrary type must be represented by meta language constructs of known type.
- Generic programs, such as *sprintf*, which work non-parametrically over values of arbitrary type.

One approach is to abandon static type checking altogether, and instead, tag every run-time value with type information to gracefully detect ill-typed code. Another is to use a more expressive static type system. For example, ad-hoc polymorphism [27], existential types [23], subtyping polymorphism [7], and set-based types [3] all help make writing some of the above programs feasible.

A third approach, when the set of types is finite and known at compile-time, is to embed values within a user-defined datatype, effectively tagging them by their type. We call such tagged values *dynamic values*. This is simple, but verbose. The responsibility of showing that a dynamic value is of an appropriate type is moved from the type checker to the programmer, who must explicitly perform case analysis on a dynamic value before its use. The type checking of dynamic values is thereby deferred from compile-time to run-time, but the type-checking algorithm is very different at the two stages.

If, however, the set of types is not known in advance, or is infinite, a *universal datatype* is required. This is typically impossible for a user to implement if the universal type embeds functions, especially in the presence of polymorphism. Existing proposals for dynamic typing [1, 17, 2] solve this problem by introducing a universal datatype of type **Dyn** as a *language primitive*, along with two operations:

- **dynamic** $t{:}\tau$, which constructs a dynamic value containing both term $t$ and a representation of its type $\tau$.
- **typecase** $d$ **of** $\{x_1 : \tau_1 \to t_1\ ; \ldots ; x_n : \tau_n \to t_n\}$, which attempts to match the type stored within dynamic value $d$ against one of $\tau_i$, binding the term in $d$ to the appropriate $x_i$, or failing gracefully if no match is found.

Using this approach *sprintf* could be written thus:[1]

$$
\begin{array}{l}
\textit{sprintf} : \mathbf{String} \to [\mathbf{Dyn}] \to \mathbf{String} \\
\textit{sprintf}\ [\,]\ [\,] = [\,] \\
\textit{sprintf}\ (\text{‘\%’} : \text{‘n’} : cs)\ (d : ds) = \\
\quad \mathbf{typecase}\ d\ \mathbf{of}\{n :\ \mathbf{Nat} \to \textit{ntostr}\ n \mathbin{+\!\!+} \textit{sprintf}\ cs\ ds\} \\
\textit{sprintf}\ (\text{‘\%’} : \text{‘b’} : cs)\ (d : ds) = \\
\quad \mathbf{typecase}\ d\ \mathbf{of}\{b :\ \mathbf{Bool} \to \textit{btostr}\ b \mathbin{+\!\!+} \textit{sprintf}\ cs\ ds\} \\
\textit{sprintf}\ (c : cs)\ ds = c : (\textit{sprintf}\ cs\ ds)
\end{array}
$$

*Research supported by ORS Award #96017029, and conducted while visiting the Oregon Graduate Institute.

†Research supported by USAF Air Material Command, contract F19628-93-C-0069, and NSF grant IRI-9625462.

‡Research conducted while visiting the Oregon Graduate Institute.



[1] We use the pattern matching of Haskell, where (‘%’ : ‘n’ : $cs$) matches any string beginning with “%n”, binding the remainder to $cs$.

and used like this:

$$\mathit{sprintf}\ \text{“\%n = \%b”}\ [\mathbf{dynamic}\ 1, \mathbf{dynamic}\ \mathbf{true}]$$

The **typecase** solution suffers two drawbacks:

- Types now live in two quite different worlds. *Static types* are generally inferred, and may be implicitly polymorphic with little added complexity for the programmer. *Dynamic types* must be mentioned explicitly within the branches of a **typecase**, and dynamic polymorphism is either forbidden [1], restricted [17], or requires the complex machinery of functors and higher order unification [2].
- Combining dynamic values together to construct a new dynamic value is tedious and verbose to write, since each constituent value requires a separate **typecase**, and the result must be wrapped by **dynamic**. §3 will demonstrate many instances where this is required.

Our main contribution is to regard dynamic typing as *staged type inference*, in which some program expressions have their type inference deferred until sufficient context is known at run-time. This approach builds directly on a general notion of staged computation [10, 26], and is free of the above problems:

- We introduce three operators for manipulating values of dynamic type, which make creating, combining and using dynamic values easy and concise (§2.2). While two of our operators correspond to **dynamic** and **typecase**, the third allows a new style of programming not possible in the typecase approach (§3).
- The same type system used at compile-time is used at run-time to decide the well-typing of dynamic values (§5.3). In particular:
  - User-written programs are free of explicit type information, and enjoy the benefits of type inference for all expressions, including those involving dynamic values (Appendix A).
  - Dynamically-typed polymorphism is implicit, and as convenient to use as statically-typed polymorphism (§3).
  - Much of the type checking of dynamic values is performed at compile-time, leaving only a residual type unification problem to be performed at run-time (§5.4).
- Even in the presence of all the above, evaluation remains sound (§5.5).

## 2 Overview

Staged computation, sometimes called run-time code generation, generates code at run-time to take advantage of run-time invariants. A staged computation produces and then executes an implementation optimised for those invariants. A contribution of this paper is to show that this can be done in a type-safe manner.

In this section we review the three operators of staged computation, extend them to stage type inference, and present a naïve operational semantics.

### 2.1 Staged Computation

Staged computation introduces three operators to construct, eliminate and combine pieces of programs. These can be used to explicitly distribute the evaluation of a program over many *run-time stages*:

- The *defer* operator, $\langle t\rangle$, defers evaluation of an expression $t$ by one stage. Writing $\mapsto\!\!\!\!\twoheadrightarrow$ for the evaluation relation, we have:

  $$\begin{array}{ll} 1+1 \mapsto\!\!\!\twoheadrightarrow 2 & \textit{evaluated at stage 0} \\ \langle 1+1\rangle \mapsto\!\!\!\twoheadrightarrow \langle 1+1\rangle & \textit{deferred till stage 1} \end{array}$$

  We call $t$ the *body* of $\langle t\rangle$, and often describe $\langle t\rangle$ as the *code* of $t$.
- The *run* operator, $\mathbf{run}(t)$, evaluates $t$ to a deferred expression $\langle u\rangle$, and then evaluates $u$. Continuing our example:

  $$\begin{array}{ll} \langle 1+1\rangle \mapsto\!\!\!\twoheadrightarrow \langle 1+1\rangle & \textit{deferred till stage 1} \\ \mathbf{run}(\langle 1+1\rangle) \mapsto\!\!\!\twoheadrightarrow 1+1 \mapsto\!\!\!\twoheadrightarrow 2 & \\ \quad \textit{evaluation brought forward to stage 0} & \end{array}$$

- The *splice* operator, $\sim t$, also evaluates $t$ to a deferred expression $\langle u\rangle$, but then splices $u$ into the body of a surrounding deferred expression. $\sim t$ is thus only legal within lexically-enclosing $\langle\rangle$ brackets. For example:

  $$\begin{array}{l} \mathbf{let}\ \mathit{code} = \langle 1+1\rangle\ \mathbf{in} \langle \sim\mathit{code} + 2\rangle \\ \quad \mapsto\!\!\!\twoheadrightarrow \langle 1+1+2\rangle \\ \qquad \sim\textit{code replaced with } 1+1 \textit{ at stage 0} \end{array}$$

  A splice expression may appear deep within the body of a deferred expression, even under a lambda abstraction:

  $$\begin{array}{l} \mathbf{let}\ \mathit{code} = \langle 1+1\rangle\ \mathbf{in} \langle \lambda y\ .\ \sim\mathit{code} + y\rangle \\ \quad \mapsto\!\!\!\twoheadrightarrow \langle \lambda y\ .\ 1+1+y\rangle \end{array}$$

  Using splice it is possible to construct and use deferred expressions that contain free variables. This is most convenient when constructing a deferred function:

  $$\begin{array}{l} \mathbf{let}\ f\ \mathit{code} = \langle \sim\mathit{code} + \sim\mathit{code}\rangle\ \mathbf{in} \langle \lambda x\ .\ \sim(f\ \langle x\rangle)\rangle \\ \quad \mapsto\!\!\!\twoheadrightarrow \langle \lambda x\ .\ x+x\rangle \\ \qquad \textit{x is free in code, but bound in final expression} \end{array}$$

An informal operational semantics for the three operators can be given by two rewrite rules. Let a subterm $u$ of $t$ be *at stage* $n$ if $u$ is nested within $n$ more $\langle\rangle$ brackets than $\sim$'s within $t$. That is:

$$\begin{array}{ll} t+u & \textit{u at stage 0} \\ \langle t+u\rangle & \textit{u at stage 1} \\ \langle t+\sim u\rangle & \textit{u at stage 0} \end{array}$$

Let $t_0$ be any term free of splice expressions, and $E$ any context such that $u$ is at the same stage as $E[u]$. Then a splice expression is rewritten by:

$$\langle E[\sim\langle t_0\rangle]\rangle \longrightarrow \langle E[t_0]\rangle$$

Now let $t_0'$ be a closed term also free of splice expressions. Then a run expression is rewritten by:

$$\mathbf{run}(\langle t_0'\rangle) \longrightarrow t_0'$$

These operators have been studied by Davies and Pfenning [10] and Taha and Sheard [26], and can be seen as generalising the work of Nielson and Nielson on two level functional languages [22]. They may be used to improve efficiency by allowing programs to be partially evaluated on-line [15].

### 2.2 Staged Type Inference

Our proposal extends these operators to stage the *type inference* of dynamic expressions along with their *evaluation.*

- $\langle t \rangle$ now defers both the type inference and evaluation of $t$ by one stage:

$$\begin{array}{ll} 1+1 : \mathbf{Nat} & \textit{inferred at compile-time} \\ 1+1 \mapsto\!\!\!\rightarrow 2 & \textit{evaluated at stage 0} \\ \\ \langle 1+1 \rangle : \langle\rangle & \textit{inference deferred} \\ \langle 1+1 \rangle \mapsto\!\!\!\rightarrow \langle 1+1 \rangle & \textit{evaluation deferred} \end{array}$$

  All deferred expressions have type $\langle\rangle$, allowing their uniform treatment by programs such as *sprintf*. The expression $\langle t \rangle$ is thus very similar to $\mathbf{dynamic}\, t$, and the type $\langle\rangle$ similar to $\mathbf{Dyn}$.

- The run operator, $\mathbf{run}(t, w)$, now takes two arguments; $t$ must evaluate to a deferred expression as before, and $w$ is an *exception expression.* $\mathbf{run}(t, w)$ evaluates $t$ to a deferred expression $\langle u \rangle$, and infers the type of $u$. If this type is compatible with the statically inferred type of $w$, then **run** evaluates $u$. For example:

$$\begin{array}{ll} \langle 1+1 \rangle : \langle\rangle & \textit{inference deferred} \\ \mathbf{run}(\langle 1+1 \rangle, 0) & \\ \quad \Longrightarrow 1+1 : \mathbf{Nat} & \\ \qquad \textit{inference brought forward to stage 0} & \\ \quad \Longrightarrow \mathbf{Nat} = \mathbf{Nat} & \textit{types are compatible} \\ \quad \mapsto\!\!\!\rightarrow 1+1 \mapsto\!\!\!\rightarrow 2 & \\ \qquad \textit{evaluation brought forward to stage 0} & \end{array}$$

  Two things can go wrong here: the type of $u$ may be incompatible with that of $w$, or $u$ may be ill-typed to begin with. If either of these occur then **run** discards $u$ and evaluates $w$ in its place. For example:

$$\begin{array}{ll} \mathbf{run}(\langle 1+1 \rangle, \mathbf{true}) & \\ \quad \Longrightarrow 1+1 : \mathbf{Nat} & \\ \qquad \textit{inference brought forward to stage 0} & \\ \quad \Longrightarrow \mathbf{Nat} \neq \mathbf{Bool} & \textit{types not compatible} \\ \quad \mapsto\!\!\!\rightarrow \mathbf{true} & \textit{exeception evaluated} \end{array}$$

  $\mathbf{run}(t, w)$ is very similar to the one branch typecase

$$\mathbf{typecase}\, t\, \mathbf{of} \{ x : \tau \to x \,;\, \mathbf{otherwise} \to w \}$$

  where $w : \tau$. We return to this at the end of this section.

- As before, $\sim t$ evaluates $t$ to $\langle u \rangle$, and splices $u$ into the body of a surrounding deferred expression. The novelty here is that the type of the resulting body depends on the type of $u$, which is only known at run-time.

  For example, if *code* yields a deferred function of type $\mathbf{Nat} \to \tau$, then the deferred expression

$$\langle (\sim code)\ 1 \rangle$$

  will have body type $\tau$. Hence, in this expression:

$$\begin{array}{l} \mathbf{let}\, code = \langle \lambda x \,.\, (x, x) \rangle\, \mathbf{in} \langle (\sim code)\ 1 \rangle \\ \quad \mapsto\!\!\!\rightarrow \langle (\lambda x \,.\, (x, x))\ 1 \rangle \end{array}$$

  we find that the resulting body has type $(\mathbf{Nat}, \mathbf{Nat})$. However, in

$$\begin{array}{l} \mathbf{let}\, code = \langle \lambda x \,.\, \mathbf{true} \rangle\, \mathbf{in} \langle (\sim code)\ 1 \rangle \\ \quad \mapsto\!\!\!\rightarrow \langle (\lambda x \,.\, \mathbf{true})\ 1 \rangle \end{array}$$

  we now find the resulting body has type **Bool**.

  It is quite possible for an expression to be incompatible with the context it is spliced into, yielding an *ill-typed deferred expression.* For example:

$$\begin{array}{l} \mathbf{let}\, code = \langle \lambda x \,.\, x+1 \rangle\, \mathbf{in} \langle (\sim code)\ \mathbf{true} \rangle \\ \quad \mapsto\!\!\!\rightarrow \langle (\lambda x \,.\, x+1)\ \mathbf{true} \rangle \end{array}$$

  Such type errors cannot, in general, be detected statically, because the static type of *code* is $\langle\rangle$. Hence, type inference must be deferred. Ill-typed dynamic expressions are detected by **run**:

$$\begin{array}{ll} \mathbf{run}(\langle (\lambda x \,.\, x+1)\ \mathbf{true} \rangle, \mathbf{false}) & \\ \quad \Longrightarrow (\lambda x \,.\, x+1)\ \mathbf{true} & \\ \qquad \textit{type inference brought forward, ill-typed} & \\ \quad \mapsto\!\!\!\rightarrow \mathbf{false} & \textit{exception evaluated} \end{array}$$

Unlike defer and **run**, splice does not correspond to any construct of previous proposals for dynamic typing based on **typecase**. In particular, the use of splice and the ability to manipulate dynamically typed code with free variables dramatically improves our power of expression over the **typecase** approach, as will be shown in §3.

We can readily mimic typecases with more than one branch using splices and nested **run**'s. For example:

$$\begin{array}{ll} \mathbf{typecase}\, t\, \mathbf{of} \{ & x : \tau_1 \to C_1[x]; \\ & x : \tau_2 \to C_2[x]; \\ & \mathbf{otherwise} \to u\ \} \end{array}$$

can be expressed as:

$$\mathbf{run}(\langle C_1[\sim t] \rangle, \mathbf{run}(\langle C_2[\sim t] \rangle, u))$$

where $C_1$ and $C_2$ are contexts, possibly with more than one hole. Notice that no explicit mention of types $\tau_1$ and $\tau_2$ was necessary.

We have emphasised how **run** uses its exception expression to determine the required type of a deferred expression. In practice, the programmer may well have an intended type in mind and would be reassured if the type checker could confirm it. For this reason a practical implementation should also allow exception expressions to be annotated by a type, and type inference should ensure the inferred type subsumes any annotated type.

### 2.3 A Naïve Semantics

We can easily refine the rewrite rules of §2.1 to give, informally, the semantics of staged type inference. The first rule remains unchanged:

$$\langle E[\sim \langle t_0 \rangle] \rangle \longrightarrow \langle E[t_0] \rangle$$

The rule for **run** must be enhanced to check that $t_0'$ is both well-typed, and matches the type of $w$:

$$\begin{array}{ll} \mathbf{run}(\langle t_0' \rangle, w) \longrightarrow & \mathbf{if} \cdot \vdash t_0' : \tau\, \mathbf{and}\, w : \tau'\, \mathbf{and}\, \tau = \tau' \\ & \mathbf{then}\, t_0'\, \mathbf{else}\, w \end{array}$$

This is done by inferring the type of $w$ at compile-time and of $t_0'$ *at run-time*, and comparing the results. Here $\cdot$ denotes the empty type context, implying that $t_0'$ must be closed. The relation $\vdash$ in the dynamic rewrite rule above *is the same relation* used in the static typing rules.

This makes a fine specification, but in practice it might be rather expensive to perform type inference on a dynamically-constructed expression at run-time. In §4 we develop an efficient implementation of this specification.

## 3 Examples

In this section we illustrate the use of our dynamic operators.

### 3.1 *sprintf* Revisited

It is straightforward to directly translate the *sprintf* example of §1 to use our constructs:

$$\begin{array}{l} \mathit{sprintf}' : \mathbf{String} \to [\langle\rangle] \to \langle\rangle \\ \mathit{sprintf}'\ [\,]\ [\,] = \langle[\,]\rangle \\ \mathit{sprintf}'\ (\text{`\%'} : \text{`n'} : \mathit{cs})\ (d : \mathit{ds}) = \\ \quad \langle \mathit{ntostr}\ {\sim}d +\!\!+ {\sim}(\mathit{sprintf}'\ \mathit{cs}\ \mathit{ds})\rangle \\ \mathit{sprintf}'\ (\text{`\%'} : \text{`b'} : \mathit{cs})\ (d : \mathit{ds}) = \\ \quad \langle \mathit{btostr}\ {\sim}d +\!\!+ {\sim}(\mathit{sprintf}'\ \mathit{cs}\ \mathit{ds})\rangle \\ \mathit{sprintf}'\ (c : \mathit{cs})\ \mathit{ds} = \langle c : {\sim}(\mathit{sprintf}'\ \mathit{cs}\ \mathit{ds}) \\ \\ \mathit{sprintf} : \mathbf{String} \to [\langle\rangle] \to \mathbf{String} \\ \mathit{sprintf}\ \mathit{cs}\ \mathit{ds} = \mathbf{run}(\mathit{sprintf}'\ \mathit{cs}\ \mathit{ds}, \\ \quad \text{“error: args mismatch format string”}) \end{array}$$

*sprintf'* traverses the format string, splicing together code to construct the result string. Once generated, *sprintf* attempts to run this code, yielding an error string should the actual argument types not agree with the format string. For example:

$$\begin{array}{l} \mathit{sprintf}\ \text{“\%n = \%b”}\ [\langle 1\rangle, \langle \mathbf{true}\rangle] \\ \quad \twoheadrightarrow \mathbf{run}(\langle \mathit{ntostr}\ 1 +\!\!+ \text{“ = ”} +\!\!+ \mathit{btostr}\ \mathbf{true} +\!\!+ [\,]\rangle, \ldots) \\ \quad \twoheadrightarrow \mathit{ntostr}\ 1 +\!\!+ \text{“ = ”} +\!\!+ \mathit{btostr}\ \mathbf{true} +\!\!+ [\,] \\ \quad \twoheadrightarrow \text{“1 = True”} \end{array}$$

Of course if the format string is a constant literal string then this traversal could occur at compile-time, and the code could be statically type checked, but in general this is not the case, so the type checking must be deferred until run-time.

We can also take advantage of our ability to manipulate code containing free variables to implement a more convenient pretty-printer which does not require its arguments to be explicitly wrapped by $\langle\rangle$ brackets:

$$\begin{array}{l} \mathit{sprintf2}' : \mathbf{String} \to \langle\rangle \to \langle\rangle \\ \mathit{sprintf2}'\ [\,]\ d = d \\ \mathit{sprintf2}'\ (\text{`\%'} : \text{`n'} : \mathit{cs})\ d = \\ \quad \langle \lambda n\ .\ {\sim}(\mathit{sprintf2}'\ \mathit{cs}\ \langle {\sim}d +\!\!+ \mathit{ntostr}\ n\rangle)\rangle \\ \mathit{sprintf2}'\ (\text{`\%'} : \text{`b'} : \mathit{cs})\ d = \\ \quad \langle \lambda b\ .\ {\sim}(\mathit{sprintf2}'\ \mathit{cs}\ \langle {\sim}d +\!\!+ \mathit{btostr}\ b\rangle)\rangle \\ \mathit{sprintf2}'\ (c : \mathit{cs})\ d = \mathit{sprintf2}'\ \mathit{cs}\ \langle {\sim}d +\!\!+ [c]\rangle \end{array}$$

*sprintf2'* traverses the format string, constructing a *function* taking all required arguments and converting them to the required string. Argument $d$ to *sprintf2'* accumulates the code to print the arguments seen so far. Notice that $n$ is free in the deferred expression passed to the recursive call to *sprintf2'*, and similarly for $b$. Without this ability it would be impossible to construct the function at run-time.

*sprintf2* attempts to run the code created by *sprintf2'*:

$$\begin{array}{l} \mathit{sprintf2} : \forall\alpha\ .\ \mathbf{String} \to \alpha \\ \mathit{sprintf2}\ \mathit{cs} = \mathbf{run}(\mathit{sprintf2}'\ \mathit{cs}\ \langle[\,]\rangle, \mathit{error}) \end{array}$$

This function has a strange type: it looks as if it can magically return any type of value. (The function *error* has type $\forall\alpha\ .\ \alpha$, and hence places no constraint on the result type.) In reality, the context surrounding each use of *sprintf2* forces it to be specialised at some particular type, and this type is dynamically tested against the value returned by *sprintf2'*. Should this test fail, the error function will be called.

Using *sprintf2* is straightforward:

$$\begin{array}{l} \mathit{sprintf2}\ \text{“\%n = \%b”}\ 1\ \mathbf{true} \\ \quad \twoheadrightarrow \mathbf{run}(\langle \lambda n\ .\ \lambda b\ .\ [\,] +\!\!+ \mathit{ntostr}\ n +\!\!+ \text{“ = ”} +\!\!+ \mathit{btostr}\ b\rangle, \\ \qquad\qquad \mathit{error})\ 1\ \mathbf{true} \\ \quad \twoheadrightarrow (\lambda n\ .\ \lambda b\ .\ [\,] + \mathit{ntostr}\ n +\!\!+ \text{“ = ”} +\!\!+ \mathit{btostr}\ b)\ 1\ \mathbf{true} \\ \quad \twoheadrightarrow \text{“1 = True”} \end{array}$$

Here, the context forces *sprintf2* to be called with $\alpha$ instantiated to $\mathbf{Nat} \to \mathbf{Bool} \to \mathbf{String}$, so this occurrence of *sprintf2* has the convenient, curried type $\mathbf{String} \to \mathbf{Nat} \to \mathbf{Bool} \to \mathbf{String}$. Compare this with the type of *sprintf*, namely $\mathbf{String} \to [\langle\rangle] \to \mathbf{String}$, which forces its arguments to be wrapped in $\langle\rangle$ brackets, and packed into a list. This is precisely the kind of expressive power the splice operator adds to the system.

This example explicitly divides evaluation into two stages, the first constructing the print function, and the second applying it. Using this strategy, a print function may be generated once, and re-applied many times, without the need to re-traverse the format string or re-type-check the resulting code.

### 3.2 Distributed Computing

The distributed programming model popularised by Java and the Microsoft Component Object Model [18] allows library code residing on potentially remote machines to be linked, at run-time, into a running program. One challenge is to ensure such distributed programs remain well-typed. We show how our approach supports type-safe distributed programming with little impact on the underlying language.

We add the primitive:

$$\mathit{getCode} : \mathbf{String} \to \mathbf{String} \to \langle\rangle$$

which given a library name and function name, returns the code representing that function. The resulting function may be implemented locally, or may reside on a remote machine.

Now suppose that library “Vector” supplies the usual vector functions (including *map*), “FFT” supplies a Fast Fourier Transform function (*fft*), and “Window” supplies some high level graphics routines (*newWindow* and *barGraph*):

$$\begin{array}{l} \mathit{map} : (\alpha \to \beta) \to \mathbf{Vector}\,\alpha \to \mathbf{Vector}\,\beta \\ \mathit{fft} : \mathbf{Nat} \to \mathbf{Vector\ Complex} \to \mathbf{Vector\ Complex} \\ \mathit{newWindow} : \mathbf{String} \to \mathbf{IO\ Window} \\ \mathit{barGraph} : \mathbf{Nat} \to \mathbf{Vector\ Real} \to \mathbf{Window} \to \mathbf{IO}() \end{array}$$

We will use the monadic IO of Haskell [24] throughout this example. Side-effecting computations which yield a value of type $\tau$ have type $\mathbf{IO}\,\tau$, and are constructed using primitive operations and the **do** construct.

Furthermore, we'll make use of two local functions for converting between scalar and complex numbers:

$$\begin{array}{l} (:+) : \mathbf{Real} \to \mathbf{Real} \to \mathbf{Complex} \\ \mathit{magnitude} : \mathbf{Complex} \to \mathbf{Real} \end{array}$$

Using these libraries we now implement a spectrograph function, which displays the frequency spectrum of a given sampled waveform in a fresh window, as follows:

$$
\begin{array}{l}
mapCode, fftCode, newWindowCode, barGraphCode : \langle\rangle \\
mapCode = getCode \text{ “Vector” “map”} \\
fftCode = getCode \text{ “FFT” “fft”} \\
newWindowCode = getCode \text{ “Window” “newWindow”} \\
barGraphCode = getCode \text{ “Window” “barGraph”}
\end{array}
$$

$$
\begin{array}{l}
spectograph : \mathbf{Nat} \rightarrow \mathbf{Vector\ Real} \rightarrow \mathbf{IO}() \\
spectograph = \\
\quad \mathbf{let}\ showSpec = \langle \lambda size\ .\ \lambda sample\ .\ \mathbf{do} \\
\qquad w \leftarrow {\sim}newWindowCode \text{ “A Spectrograph”} \\
\qquad \mathbf{let}\ freq = {\sim}fftCode\ size\ ({\sim}mapCode\ (:+\ 0)\ sample) \\
\qquad \mathbf{let}\ mag = {\sim}mapCode\ magnitude\ freq \\
\qquad {\sim}barGraphCode\ size\ mag\ w\rangle \\
\quad \mathbf{in}\ \mathbf{let}\ errorSpec = \lambda\_\ .\ \lambda\_\ .\ userError \text{ “build failed”} \\
\quad \mathbf{in}\ \mathbf{run}(showSpec, errorSpec)
\end{array}
$$

Notice firstly that it is almost as convenient to use a dynamically-bound function (such as *mapCode* or *fftCode*) as an ordinary statically-bound function (such as $(:+)$ or *magnitude*). All that is required is a single call to *getCode*, and a splice at each call site. In particular, a single call to *getCode* can return a *polymorphic* function. For example, *mapCode* is used at two different call sites, each at a different vector type.

Secondly, notice that all type checking is localised to the **run** within *spectrograph*, and performed (at most) once. In particular, once the body of *spectrograph* has been evaluated, each call to the resulting function performs no dynamic type checks.

How could *getCode* be implemented? If asked for a function which is local, the returned code need only redirect the call:

$$getCode \text{ “FFT” “fft”} \mapsto\!\!\!\!\rightarrow \langle FFT\,.\,fft\rangle$$

Then calls to local functions are without performance penalty.

Should the requested function resides on a remote machine, then there are two possibilities. The first is for *getCode* to return the code of the function itself, which may then be run locally:

$$getCode \text{ “FFT” “fft”} \mapsto\!\!\!\!\rightarrow \langle \lambda size\ .\ \lambda vector\ .\ body\ of\ fft\rangle$$

The second is for *getCode* to use a new primitive, *rpc*, to generate code to perform a remote call:

$$rpc : \mathbf{String} \rightarrow \mathbf{String} \rightarrow \mathbf{String} \rightarrow [\langle\rangle] \rightarrow \langle\rangle$$

Given a machine name, library name, function name and list of arguments, *rpc* returns the code required to call the given function, taking care to establish a network connection, marshal all the arguments, and finally unmarshal the result:

$$
\begin{array}{l}
getCode \text{ “FFT” “fft”} \mapsto\!\!\!\!\rightarrow \langle \lambda size\ .\ \lambda vector\ . \\
\quad {\sim}(rpc \text{ “MachineName” “FFT” “fft” } [\langle size\rangle, \langle vector\rangle])\rangle
\end{array}
$$

## 4 A Smarter Semantics: Incremental Staged Type Inference

The naïve semantics of §2.3 defers all type inference to **run**. While this is an appealingly simple specification, it is not a very good basis for an efficient implementation:

- If $\mathbf{run}(\langle t\rangle, u)$ is to decide $\cdot \vdash t : \tau$ at run-time, then $t$ must be represented at run-time in an abstract form amenable to type inference, rather than a compiled form amenable to efficient execution.
- $t$ will be type checked afresh every time it is re-run.
- A subterm spliced into $t$ more than once will also be type checked afresh at every occurrence.
- Every subterm of $t$ will be type checked, even those to which a type could have been assigned at compile-time.

We now sketch a more efficient semantics motivated by an analogy with the standard way of typing let-bound polymorphism.

A possible *specification* for type checking polymorphic let expressions is to first unfold the let-bound term into the let body, and then type check the resulting let body. However, it is not hard to see that the resulting type derivation will contain duplicated sub-derivations for each occurrence of the let-bound term. An *implementation* will instead type check the let-bound term, *generalise* its type to give a type scheme, and then *specialise* the type scheme at each use within the let body. We can imagine the type scheme as a *suspended type derivation* which is restarted at each use. This technique moves much of the work of type checking the let-bound expression from its many call sites, to its unique binding site, leaving only a (small) residual problem at each call site.

A similar implementation technique is also appropriate for deferred expressions. The naïve semantics type checks a deferred expression only after every splice within it has been unfolded. If the same deferred expression were spliced many times, the type derivation would contain duplicated sub-derivations. To avoid this we introduce *dynamic type schemes* to represent, *at run-time*, the suspended type derivation of a deferred expression. This derivation will terminate at each splice expression, pending further type information.

Unlike type schemes, dynamic type schemes are not made visible in the type of the deferred expression, which remains as $\langle\rangle$. Instead, this type information is stored inside the *run-time value* representing the deferred expression in the form of *type annotations*. For example:

$$\langle \alpha, \beta{:}\overline{*} \vdash \lambda x{:}\alpha\ .\ (x, ({\sim}f{:}\alpha \rightarrow \beta)\ x) : (\alpha, \beta)\rangle$$

is an annotated deferred expression, and will be explained in §5. This notation for annotated deferred expressions is intended to strengthen our intuition of a dynamic type scheme as a suspended type derivation.

The operational semantics exploits the type annotations to keep track of the type inside a deferred expression as each splice expression within it is executed. When a deferred expression $\langle t\rangle$ is spliced into another deferred expression $\langle u\rangle$ at run-time, the suspended derivation in $\langle u\rangle$ is restarted, and “combined” with the suspended derivation in $\langle t\rangle$. The beauty of this approach is that this “combination” is simply a matter of unifying two types, and applying the resulting substitution to $t$ and $u$.

How does this work? The first step is to annotate, at compile-time, each splice expression with the most general type compatible with its context. This type may contain free type variables at points where its type is unconstrained. We also annotate each deferred expression with the type of its body. This type may mention the type variables appearing in splice expressions, so capturing the dependency of its type on what is spliced into its body. Finally, each **run** is annotated with the type of its exception expression.

Then at run-time, if when evaluating a deferred expression we come across $\sim\langle t\rangle$, we first check that $t$ is type compatible with its context by unifying the required type (annotated on the splice) with the actual type (annotated on $t$). If this succeeds, we replace $\sim\langle t\rangle$ with $t$, and update the deferred expression we are evaluating to incorporate the actual type of $t$ by applying the unifying substitution to it.

If these types are incompatible, we will mark the deferred expression as *ill-typed*, which will be detected by splice and **run**.

Now, when evaluating $\mathbf{run}(\langle t\rangle, u)$, all that is left to do is to check that the type of $t$ can be unified with the type of $u$. This is very easy, as $t$ has maintained its type throughout evaluation, and $u$'s type is annotated on the **run** itself. Of course, should **run** find a failed deferred expression, it will evaluate its exception expression immediately.

To illustrate, we return to an example of §2.2, modified to also include a **run**:

$$\mathbf{let}\ \mathit{code} = \langle \lambda x\ .\ (x, x)\rangle\ \mathbf{in}\ \mathbf{run}(\langle(\sim\mathit{code})\ 1\rangle, (0,0))$$

At compile-time, type inference for the **let**-bound deferred expression assigns its body the type $\alpha \to (\alpha, \alpha)$ for free type variable $\alpha$. Similarly, type inference tells us *code* must yield a $\mathbf{Nat} \to \beta$ function, for free type variable $\beta$, and that the resulting deferred expression has type $\beta$. Type inference for the **run** tells us it expects a deferred expression with body type $(\mathbf{Nat}, \mathbf{Nat})$.

We simply record all of this information in the term as (roughly):

$$\begin{array}{l}\mathbf{let}\ \mathit{code} = \langle \lambda x\ .\ (x, x) : \alpha \to (\alpha, \alpha)\rangle \\ \quad \mathbf{in}\ \mathbf{run}(\langle(\sim\mathit{code}{:}\mathbf{Nat} \to \beta)\ 1 : \beta\rangle, (0,0){:}(\mathbf{Nat}, \mathbf{Nat}))\end{array}$$

At run-time we first wish to splice *code* into its context, so we unify

$$\alpha \to (\alpha, \alpha) = \mathbf{Nat} \to \beta$$

to give the substitution

$$[\alpha \mapsto \mathbf{Nat}, \beta \mapsto (\mathbf{Nat}, \mathbf{Nat})]$$

We then perform the splice, yielding

$$\langle(\lambda x\ .\ (x, x))\ 1 : \beta\rangle$$

to which we apply the above substitution to give

$$\langle(\lambda x\ .\ (x, x))\ 1 : (\mathbf{Nat}, \mathbf{Nat})\rangle$$

thus incorporating the type of the body of *code* into the overall deferred expression type.

Finally, **run** checks for type compatibility by unifying

$$(\mathbf{Nat}, \mathbf{Nat}) = (\mathbf{Nat}, \mathbf{Nat})$$

and, since this trivially succeeds, continues to evaluate

$$(\lambda x\ .\ (x, x))\ 1$$

to give our result

$$(1, 1)$$

What distinguishes a dynamic type scheme from a conventional type scheme is the use of unification rather than type specialisation at each splice point. Looking again at our previous example, we see that making sure $\mathbf{Nat} \to \beta$ was compatible with $\alpha \to (\alpha, \alpha)$ required unification because both *code* and the defer expression it was spliced into constrain each other's types. Intuitively, *type information flows both ways across a splice*, whereas it flows in only one direction in type specialisation.

In the next section we will develop the type system, type annotations and operational semantics necessary to make all of this formal. We will see that our system performs most type inference at compile-time, and distributes the residual type checking at run-time across each splice and **run** expression. This has many advantages:

- Run-time type processing is only required at splice and **run** points in the program, does not depend on the syntax of the expression in which they appear or the syntax of the expressions they contain, and requires only simple type unification.
- It is thus possible to represent $\langle t\rangle$ by a compiled form of $t$ together with some additional type-related information, rather than as an abstract representation of $t$.
- The same deferred expression may be run multiple times without duplicating type inference.
- Similarly, the same deferred expression may be spliced multiple times without duplicating type inference.
- Most importantly, polymorphism is handled very naturally.

## 5 Formal Development

We now formalise our system as $\lambda\mathsf{dyn}$, a small call-by-value lambda-calculus extended with dynamic constructs.

### 5.1 Syntax

*Source terms*, defined in Figure 1, are implicitly typed. As usual, **let** introduces polymorphic terms, and **letrec** is syntactically restricted to bind a lambda term. Most of our examples assume $\lambda\mathsf{dyn}$ to be extended with naturals, case, booleans, pairs and other constructs in the obvious way.

Type annotation (Appendix A) reconstructs the types of implicitly typed source terms and rewrites them to explicitly typed *annotated terms*, defined in Figure 2. Most of these annotations are exploited by the operational semantics of §5.4.

Types include the empty type $\mathbf{0}$, function spaces, and $\langle\rangle$, the universal type shared by all dynamic expressions. We take the *type scheme* $\forall\overline{\alpha}\ .\ \tau$ to be the type $\tau$ in the case $\overline{\alpha}$ is $\cdot$, the empty type vector.

The syntax of annotated terms is complicated slightly by the need to syntactically restrict let-bound terms to *splice-free* terms. This restriction will be justified in §6.1. A term is splice-free if each of its subterms is at a stage $\geq 0$ (see §2.1). For example:

$$\begin{array}{ll}\sim\mathit{code} + 1 & \textit{splice-full} \\ \langle\sim\mathit{code} + 1\rangle & \textit{splice-free} \\ \langle\sim(\sim\mathit{code}) + 1\rangle & \textit{splice-full}\end{array}$$

| | | |
|---|---|---|
| Variables | $x, y, z$ | |
| Constants | $c$ | |
| Source terms | $e, f, g$ | $::= \ x \mid c \mid \lambda x\ .\ e \mid e\ f \mid \ \mathbf{let}\, x = e\, \mathbf{in}\, f \mid \ \mathbf{letrec}\, x = \lambda y\ .\ e\, \mathbf{in}\, f \mid \langle e \rangle \mid \sim e \mid \mathbf{run}(e, f)$ |

Figure 1: Syntax of λdyn source terms

| | | |
|---|---|---|
| Type variables | $\alpha, \beta$ | |
| Types | $\tau, \upsilon$ | $::= \ \alpha \mid \mathbf{0} \mid \tau \to \upsilon \mid \langle\rangle$ |
| Type schemes | $\sigma, \phi$ | $::= \ \forall \overline{\alpha}\ .\ \tau$ |
| Annotated terms | $t, u, w$ | $::= \ x \mid c \mid \lambda x{:}\tau\ .\ t \mid t\ u \mid \lambda \overline{\alpha}{:}\overline{*}\ .\ t \mid t\ \overline{\tau} \mid \ \mathbf{let}\, x{:}\sigma = t_0\, \mathbf{in}\, u$ |
| | | $\mid \ \mathbf{letrec}\, x{:}\tau = \lambda y{:}\upsilon\ .\ t\, \mathbf{in}\, u \mid \langle \overline{\alpha}{:}\overline{*} \vdash t : \tau \rangle \mid \langle \mathbf{Fail} \rangle \mid \sim t{:}\tau \mid \mathbf{run}(t, u{:}\tau)$ |
| Splice free terms | $t_0, u_0, w_0$ | $::= \ t$ such that $\mathit{spliceFree}(t)$ |

Figure 2: Syntax of λdyn types and annotated terms

| | | |
|---|---|---|
| $\mathit{spliceFree}(t)$ | $=$ | $\mathit{minStage}(t) \geq 0$ |
| $\mathit{minStage}(x)$ | $=$ | $0$ |
| $\mathit{minStage}(\langle \overline{\alpha}{:}\overline{*} \vdash t : \tau \rangle)$ | $=$ | $\mathit{minStage}(t) + 1$ |
| $\mathit{minStage}(\langle \mathbf{Fail} \rangle)$ | $=$ | $0$ |
| $\mathit{minStage}(\sim t{:}\tau)$ | $=$ | $\mathit{minStage}(t) - 1$ |
| $\mathit{minStage}(\mathbf{run}(t, u{:}\tau))$ | $=$ | $\mathit{min}\ \mathit{minStage}(t)\ \mathit{minStage}(u)$ |

Figure 3: The *spliceFree* predicate (extract)

This condition is formalised by the predicate *spliceFree*, defined in Figure 3, which uses the auxiliary function *minStage* to walk over a term, keeping count of the number of defer and splice expressions entered. We let $t_0$ (and $u_0$, $w_0$) range over splice-free terms.

Type annotations fall into four groups:

- Lambda bound variables are annotated by their type so that we may use a Church-style presentation of our type system (§5.3), but otherwise serve no purpose at run-time. Thus $\lambda x\ .\ x + 1$ is annotated as $\lambda x{:}\,\mathbf{Nat}\ .\ x + 1$.
- We make all type generalisation and specialisation explicit using the type abstraction and application of System F [12]. For example, the term

  $$\mathbf{let}\, f\ x\ y = x\, \mathbf{in}(f\ 1\ \ \mathbf{true}, f\ 2\ 3)$$

  would be annotated as

  $$\begin{aligned}&\mathbf{let}\, f{:}\forall \alpha, \beta\ .\ \alpha \to \beta \to \alpha = \lambda \alpha, \beta{:}\overline{*}\ .\ \lambda x{:}\alpha\ .\ \lambda y{:}\beta\ .\ x\\ &\quad \mathbf{in}(f\ \mathbf{Nat}\ \mathbf{Bool}\ 1\ \mathbf{true}, f\ \mathbf{Nat}\ \mathbf{Nat}\ 2\ 3)\end{aligned}$$

  As in pure type systems [5] we use $\lambda\alpha : *\ .\ t$ for type abstraction instead of the usual $\Lambda\alpha\ .\ t$, and use $*$ to denote the kind of all types. For convenience, we also allow multiple type variables to be bound by a single lambda, writing $\lambda\alpha, \beta : \overline{*}\ .\ t$. Here $\overline{*}$ denotes (informally) the kind of all type vectors.
- $\mathbf{run}(t, u)$ is annotated by the most general type of its context. For example:

  $$(\mathbf{run}(t, \lambda x\ .\ x)\ 1) + 1$$

  is annotated as

  $$(\mathbf{run}(t, (\lambda x{:}\,\mathbf{Nat}\ .\ x){:}\,\mathbf{Nat} \to \mathbf{Nat})\ 1) + 1$$
- A defer expression $\langle t \rangle$ is annotated with its dynamic type scheme, made up of three components:
  - Each splice expression within $t$ is annotated by the most general type required by its context. Thus

    $$\langle \sim f\ 1 \rangle$$

    becomes

    $$\langle (\sim f{:}\,\mathbf{Nat} \to \alpha)\ 1 \rangle$$

    The type variable $\alpha$ is a placeholder which will be filled when the body of $f$ is spliced in during execution.
  - $t$ itself is annotated by the most general type inferred for its body. This type may depend on the types of lambda bound variables within $t$, for example:

    $$\langle (\lambda x{:}\alpha\ .\ x) : \alpha \to \alpha \rangle$$

    It may also depend on the types of splice expressions within $t$, for example:

    $$\langle (\sim f{:}\,\mathbf{Nat} \to \alpha)\ 1 : \alpha \rangle$$

    And of course, it may depend on both:

    $$\langle \lambda x{:}\alpha\ .\ (x, (\sim f{:}\alpha \to \beta)\ x) : (\alpha, \beta) \rangle$$
  - The free type variables introduced in the above annotations are listed in the *type variable list* of the defer expression, and considered bound by it. The previous example is fully annotated as:

    $$\langle \alpha, \beta{:}\overline{*} \vdash \lambda x{:}\alpha\ .\ (x, (\sim f{:}\alpha \to \beta)\ x) : (\alpha, \beta) \rangle$$

$\langle \mathbf{Fail} \rangle$ denotes an ill-typed deferred expression.

We assume all bound variables are distinct from the free variables of their context, and will take care to maintain this invariant during reduction.

| | | | |
|---|---|---|---|
| Stage numbers | $n, m$ | $\in$ | $\mathbb{N}$ |
| Type contexts | $\Gamma$ | $::=$ | $\Gamma, \overline{\alpha} : (\overline{*}, n)$ |
| | | $\mid$ | $\Gamma, x : (\sigma, n) \mid \cdot$ |
| Unit term substitutions | $\rho$ | $::=$ | $[x \mapsto t]$ |
| Unit type substitutions | $\theta$ | $::=$ | $[\alpha \mapsto \tau]$ |
| Type substitutions | $\Theta$ | $::=$ | $\theta_n \circ \ldots \circ \theta_1 \mid \mathbf{Id} \mid \mathbf{Fail}$ |

Figure 4: Syntax of $\lambda$dyn substitutions and type contexts

### 5.2 Type Contexts and Substitutions

Type checking must keep track of the stage at which each free variable is bound. To this end a *type context*, $\Gamma$, is defined in Figure 4 as an ordered finite map from term variables to both their type and the stage of the lambda-abstraction they were bound in. As type variables may also be bound at arbitrary stages, we let $\Gamma$ also map type variables to their kind and stage, much as in pure type systems [5]. We abbreviate sequences of type variable bindings within $\Gamma$ by a single type variable vector binding.

We write $ftv(\Gamma)$ for the set of free type variables appearing within the types of term variables in $\Gamma$, union the set of type variables themselves in the domain of $\Gamma$. We also write $fv(t)$ ($ftv(t)$) for all the free variables (free type variables only) in term $t$, and similarly for types.

Figure 4 also defines the syntax of substitutions. Name capture avoiding unit substitution on terms and types is applied postfix. It is defined much as in Barendregt [6], and takes care to rename the bound variables in lambda abstractions, type abstractions, **let** and **letrec** expressions, and the type variable list of deferred expressions. This will be important in §5.4.2.

Unit type substitution is also naturally extended to type contexts.

A (non-unit) *type substitution*, $\Theta$, also in Figure 4, is a composition of unit type substitutions, or the special substitutions **Id** or **Fail**. Type substitutions are applied prefix. **Id** is the everywhere identity map, where $\mathbf{Id}\ t \equiv t$, and **Fail** is the failure substitution whose action on types and terms is undefined. Let $dom(\Theta)$ denote the set of $\alpha$ within the substitutions $[\alpha \mapsto \tau]$ defining $\Theta$.

We will often abbreviate $[\alpha_n \mapsto \tau_n] \circ \ldots \circ [\alpha_1 \mapsto \tau_1]$ as $[\overline{\alpha} \mapsto \overline{\tau}]$; however we emphasise this notation does *not* denote parallel substitution, which we do not require.

We let $unify_{\overline{\alpha}}(\tau, \tau')$ be the most general idempotent substitution $\Theta$ such that $dom(\Theta) \subseteq \overline{\alpha}$ and $\Theta\tau = \Theta\tau'$, or **Fail** if no such unifier exists. If $\overline{\alpha}$ is unspecified then the substitution domain is unconstrained.

We write $\Theta_1 \sqcup \Theta_2$ to denote substitution $\Theta$ such that, for all $\alpha$

$$\Theta\alpha = unify(\Theta_1\alpha, \Theta_2\alpha)(\Theta_1\alpha)$$

### 5.3 Type Checking

The source program is processed at compile-time by the type *reconstruction* algorithm. It infers types and translates the source program to the language of annotated terms. Both the algorithm and the translation are quite standard, and are given in Appendix A.

In this section we concentrate on the type *checking* algorithm, that checks whether an annotated term is well typed. This judgement is used (i) in our proof of subject reduction, to show that a well-typed program remains well typed during reduction, and (ii) to underpin the run-time type checks needed by **run** and splice.

Figure 5 presents rules for deciding the well typing of an annotated term. We intend the judgement

$$\Gamma \vdash_n t : \sigma$$

to be true when term $t$ at stage $n$ has type $\sigma$ in type context $\Gamma$. This judgement describes well typed static terms, but is also the basis for deciding the well-typing of dynamic terms. Importantly, the very same judgement is used in both cases.

Most rules are those of System F parameterised by stage numbers.

Rule var enforces *binding time correctness* by preventing access to a variable before it is bound. Thus

$$\langle \alpha{:}* \vdash \lambda x{:}\langle\rangle\ .\ \text{\textasciitilde}x : \alpha \rangle$$

where $x$ is bound at stage 1 but accessed at stage 0, is rejected. To enforce this the type rules keep track of the stage of each subterm. This rule also supports *cross stage persistence* by allowing a variable bound at an earlier stage to be used at a later stage without coercion. Thus

$$\lambda x{:}\,\mathbf{Nat}\ .\ \langle \cdot \vdash x : \mathbf{Nat} \rangle$$

is accepted.

Rule let has the side condition $spliceFree(t)$, which we discuss in §6.1.

There are two possibilities when type checking deferred expressions. Rule defer requires $t$ to have the type it is annotated with, but at the next stage. The bound type variables $\overline{\alpha}$ are introduced into $\Gamma$. Rule fail handles the case when a deferred expression's body no longer has a valid type, thanks to splicing an expression with incompatible type. However, the deferred expression itself remains well typed as $\langle\rangle$.

Rule splice requires $t$ to yield a deferred expression at the previous stage, and $\tau$, the type required by its context, to be well formed. The auxiliary judgement

$$\Gamma \vdash_n \sigma : *$$

holds when $\sigma$ is a well-formed type scheme at stage $n$ in type context $\Gamma$. Type variables must be in scope, and be binding time correct exactly as for term variables. Its definition is straightforward.

Rule run ensures $t$ yields a defer expression, and the exception expression $u$ has a type consistent with the run expression's context.

### 5.4 Operational Semantics

Following Wright and Felleisen [28] we present our operational semantics as a rewrite system with explicit redex contexts. This small-step semantics is entirely syntactical and simplifies our proof of type soundness.

We distinguish a subset of terms to be *values*, shown in Figure 6. A defer expression with a splice-free body $t_0$ is considered a value analogously to a lambda abstraction, which must also be splice-free.

$$\frac{m \leq n}{\Gamma, x : (\sigma, m) \vdash_n x : \sigma}\;\textsf{var} \qquad \frac{\mathit{typeOf}(c) = \tau}{\Gamma \vdash_n c : \tau}\;\textsf{const}$$

$$\frac{\Gamma \vdash_n \tau : * \quad \Gamma, x : (\tau, n) \vdash_n t : \upsilon}{\Gamma \vdash_n \lambda x{:}\tau \,.\, t : \tau \to \upsilon}\;\textsf{abs} \qquad \frac{\Gamma \vdash_n t : \tau \to \upsilon \quad \Gamma \vdash_n u : \tau}{\Gamma \vdash_n t\ u : \upsilon}\;\textsf{app}$$

$$\frac{\Gamma, \overline{\alpha} : (\overline{*}, n) \vdash_n t : \tau}{\Gamma \vdash_n \lambda\overline{\alpha}{:}\overline{*} \,.\, t : \forall\overline{\alpha} \,.\, \tau}\;\textsf{Abs} \qquad \frac{\Gamma \vdash_n \overline{\tau} : \overline{*} \quad \Gamma \vdash_n t : \forall\overline{\alpha} \,.\, \upsilon}{\Gamma \vdash_n t\ \overline{\tau} : \upsilon[\overline{\alpha} \mapsto \overline{\tau}]}\;\textsf{App}$$

$$\frac{\Gamma \vdash_n t : \sigma \quad \Gamma, x : (\sigma, n) \vdash_n u : \tau \quad \mathit{spliceFree}(t)}{\Gamma \vdash_n \ \mathbf{let}\, x{:}\sigma = t\, \mathbf{in}\, u : \tau}\;\textsf{let} \qquad \frac{\Gamma, x : (\tau, n) \vdash_n t : \tau \quad \Gamma, x : (\tau, n) \vdash_n u : \upsilon}{\Gamma \vdash_n \ \mathbf{letrec}\, x{:}\tau = t\, \mathbf{in}\, u : \upsilon}\;\textsf{letrec}$$

$$\frac{\Gamma, \overline{\alpha} : (\overline{*}, n+1) \vdash_{n+1} t : \tau}{\Gamma \vdash_n \langle \overline{\alpha}{:}\overline{*} \vdash t : \tau \rangle : \langle\rangle}\;\textsf{defer} \qquad \frac{}{\Gamma \vdash_n \langle \mathbf{Fail} \rangle : \langle\rangle}\;\textsf{fail}$$

$$\frac{\Gamma \vdash_{n+1} \tau : * \quad \Gamma \vdash_n t : \langle\rangle}{\Gamma \vdash_{n+1} ({\sim}t{:}\tau) : \tau}\;\textsf{splice} \qquad \frac{\Gamma \vdash_n t : \langle\rangle \quad \Gamma \vdash_n u : \tau}{\Gamma \vdash_n \mathbf{run}(t, u{:}\tau) : \tau}\;\textsf{run}$$

Figure 5: Type checking λdyn terms

$$\begin{array}{llcl}
\text{Ground types} & \tau_0, \upsilon_0 & ::= & \tau \quad \text{such that } \mathit{ftv}(\tau) = \emptyset \\
\text{Values} & v & ::= & x \mid c \mid \lambda x{:}\tau \,.\, t_0 \mid \lambda\overline{\alpha}{:}\overline{*} \,.\, t_0 \mid \langle \overline{\alpha}{:}\overline{*} \vdash t_0 : \tau \rangle \mid \langle \mathbf{Fail} \rangle \\
\text{Stage 0 contexts} & E_0 & ::= & [] \mid E_0\ u_0 \mid v\ E_0 \mid E_0\ \overline{\tau} \mid \ \mathbf{let}\, x{:}\sigma = E_0\, \mathbf{in}\, u_0 \\
 & & \mid & \langle \overline{\alpha}{:}\overline{*} \vdash E_1[{\sim}E_0{:}\tau] : \tau' \rangle \mid \mathbf{run}(E_0, u_0{:}\tau) \\
\text{Stage} \geq 1 \text{ contexts} & E_1 & ::= & [] \mid \lambda x{:}\tau \,.\, E_1 \mid E_1\ u \mid t\ E_1 \mid E_1\ \overline{\tau} \mid \ \mathbf{let}\, x{:}\sigma = t_0\, \mathbf{in}\, E_1 \\
 & & \mid & \mathbf{letrec}\, x{:}\tau = E_1\, \mathbf{in}\, u \mid \ \mathbf{letrec}\, x{:}\tau = v\, \mathbf{in}\, E_1 \\
 & & \mid & \langle \overline{\alpha}{:}\overline{*} \vdash E_1[{\sim}E_1{:}\tau] : \tau' \rangle \mid \mathbf{run}(E_1, u{:}\tau) \mid \mathbf{run}(t, E_1{:}\tau)
\end{array}$$

Figure 6: Syntax of λdyn ground types, values and rewrite contexts

#### 5.4.1 Contexts

The call-by-value semantics of application, and the semantics of defer, splice and run, make use of two contexts $E_0$ and $E_1$, also defined in Figure 6. A context is a special term with a single "hole", denoted in the syntax by []. We write $C[t]$ to denote the context $C$ with hole filled by $t$. Note that this operation *does not* perform alpha-conversion on $C$ or $t$.

Context $E_0$, to be used when rewriting terms to values, allows a hole to appear anywhere provided:

(i) it is at stage 0,

(ii) it is not underneath a type or term lambda abstraction, let binding or letrec binding, and

(iii) all subterms at stage 0 to its left are already values.

Similarly, context $E_1$, to be used when rewriting terms to splice-free terms, allows a hole to appear anywhere provided the hole is at the same stage as the context. Note that $E_1$ *does* allow a hole to appear under a lambda abstraction, and within equal numbers of nested splice and defer expressions. However, it *does not* allow a hole to appear under a type variable abstraction. Indeed, thanks to our syntactic restriction on **let** bound terms this form of context is never required during rewriting.

We extend type and term substitution to rewrite contexts in the obvious way, where $\Theta[] = []$.

#### 5.4.2 Notions of Reduction

The *reduction relation*, $\longrightarrow$, is defined by the rules ("notions of reduction") of Figure 7. Constants, which must be of functional type, are rewritten according to the *delta function* $\delta$, which must obey the *delta constraint*

$$\begin{array}{c} \mathit{typeOf}(c) = \tau \to \upsilon \text{ and } \Gamma \vdash_n v : \tau \text{ implies} \\ \mathit{delta}(c, v) \text{ is defined and } \Gamma \vdash_n \delta(c, v) : \upsilon \end{array}$$

Rules for $\beta$ reduction of term abstractions, type abstractions, let and letrec expressions are standard. The call-by-value semantics is enforced by only allowing values to be substituted into a term.

Unlike the call-by-value and call-by-name lambda-calculus, a value $v$ may be a deferred expression containing free type or term variables, and so care must be taken to avoid free

$$
\begin{array}{llcll}
\delta & c\ v & \longrightarrow & \delta(c, v) & \text{if } \delta(c, v) \text{ defined} \\
\beta_{\mathsf{v}} & (\lambda x{:}\tau\ .\ t_0)\ v & \longrightarrow & t_0[x \mapsto v] & \\
\beta_{\tau} & (\lambda \overline{\alpha}{:}\overline{*}\ .\ t_0)\ \overline{\tau_0} & \longrightarrow & t_0[\overline{\alpha} \mapsto \overline{\tau_0}] & \text{if } \mathit{length}(\overline{\alpha}) = \mathit{length}(\overline{\tau_0}) \\
\mathsf{let} & \mathbf{let}\, x{:}\sigma = v\, \mathbf{in}\, t_0 & \longrightarrow & t_0[x \mapsto v] & \\
\mathsf{letrec} & \mathbf{letrec}\, x{:}\tau = v\, \mathbf{in}\, t_0 & \longrightarrow & t_0[x \mapsto \mathbf{letrec}\, x{:}\tau = v\, \mathbf{in}\, v] & \\
\mathsf{splice} & \langle \overline{\alpha}_1{:}\overline{*} \vdash E_1[\sim\langle \overline{\alpha}_2{:}\overline{*} \vdash t_0{:}\tau_2\rangle{:}\tau_2'] : \tau_1\rangle & \longrightarrow & \mathbf{if}\ \Theta \neq \mathbf{Fail}\ \mathbf{then} \langle \overline{\beta}{:}\overline{*} \vdash \Theta E_1[t_0] : \Theta\tau_1\rangle\, \mathbf{else} \langle \mathbf{Fail}\rangle & \\
 & & & \text{where } \Theta = \mathit{unify}_{(\overline{\alpha}_1, \overline{\alpha}_2)}(\tau_2, \tau_2') & \\
 & & & \text{and } \overline{\beta} = (\overline{\alpha}_1, \overline{\alpha}_2) \setminus \mathit{dom}(\Theta) & \\
\mathsf{splice/fail} & \langle \overline{\alpha}{:}\overline{*} \vdash E_1[\sim\langle \mathbf{Fail}\rangle{:}\tau'] : \tau\rangle & \longrightarrow & \langle \mathbf{Fail}\rangle & \\
\mathsf{run} & \mathbf{run}(\langle \overline{\alpha}{:}\overline{*} \vdash t_0 : \tau\rangle, u_0{:}\tau_0') & \longrightarrow & \mathbf{if}\ \mathit{fv}(t_0) \subseteq \overline{\alpha}\, \mathbf{and}\, \Theta \neq \mathbf{Fail}\ \mathbf{then}\, \Theta t_0[\overline{\beta} \mapsto \overline{\mathbf{0}}]\, \mathbf{else}\, u_0 & \\
 & & & \text{where } \Theta = \mathit{unify}_{\overline{\alpha}}(\tau, \tau_0') & \\
 & & & \text{and } \overline{\beta} = \overline{\alpha} \setminus \mathit{dom}(\Theta) & \\
\mathsf{run/fail} & \mathbf{run}(\langle \mathbf{Fail}\rangle, u_0{:}\tau_0) & \longrightarrow & u_0 &
\end{array}
$$

Figure 7: Rewriting $\lambda$dyn annotated terms

variable capture in substitution. For example:

$$
\begin{array}{ll}
\mathbf{let}\, f\ \mathit{code} = \langle \lambda x\ .\ \sim\mathit{code} + x\rangle\, \mathbf{in} \langle \lambda x\ .\ \sim(f\ \langle x\rangle)\rangle & \\
\mapsto\!\!\!\!\rightarrow \langle \lambda x\ .\ \lambda y\ .\ x + y\rangle & \textit{rename inner } x
\end{array}
$$

Thus $t[x \mapsto v]$ must rename bound type and term variables in $t$. This renaming would be an issue for a real implementation of our semantics.

Rule splice shows how one defer expression may be spliced into another. It uses the context $E_1$ to allow a splice expression to be rewritten anywhere within a defer expression, provided it is at stage 1.

At the term level rule splice is very simple — it simply replaces the entire splice expression with the body of the inner expression, $t$. At the type level things are slightly more complicated. It must first check that the type of the inner defer, $\tau_2$, matches that required by its context, $\tau_2'$, by unifying these types relative to the free type variables in $\overline{\alpha}_1$ and $\overline{\alpha}_2$. In practice, because we are using call-by-value, every free type variable within $\tau_2$ and $\tau_2'$ will be within $\overline{\alpha}_1$ and $\overline{\alpha}_2$, and so this constraint on unification may be safely ignored.

If this unification succeeds, rule splice applies the resulting substitution to the entire deferred expression, including all the type annotations within it. It also discards any type variables just bound, storing the remainder in the result deferred expression.

If this unification fails then the deferred expression is no longer well-typed, and is rewritten to $\langle \mathbf{Fail}\rangle$.

An actual implementation would most likely represent free type variables by references to types, and thus the application of $\Theta$ to $E_1[t]$ and $\tau$ in the above would have been done as $\Theta$ was constructed.

Rule splice/fail catches an attempt to splice an ill-typed deferred expression into another. The failure is simply propagated to the outer defer, which is rewritten to $\langle \mathbf{Fail}\rangle$.

Rule run allows a deferred expression to be brought forward and evaluated immediately. This rule is restricted both in its left-hand and right-hand sides. On the left-hand side, the rule is only applicable when (i) the body of the defer expression, $t_0$, has already been fully expanded, *i.e.*, is splice-free; and (ii) the required type $\tau_0'$ is ground. The reason for the first requirement is obvious: splice expressions are only legal within $\langle\rangle$ brackets, and thus must be fully evaluated before these brackets can be removed by the **run**. The second requirement is more subtle, and prevents free type variables escaping from the deferred expression body. However, again thanks to call-by-value, this restriction will always hold.

On the right-hand side, rule run first makes a run-time check that $t_0$ is closed. This test is essential to preserve type soundness, will be motivated in §6.2. If $t_0$ is closed, the rule then attempts to unify $t_0$'s type $\tau$ with the type expected by **run**'s context $\tau_0'$. If successful, this unifying substitution is applied to $t_0$ to give the rewritten term. If either of the above tests failed, the result is the exception expression $u$.

There is one final subtlety with **run**. Consider:

$$\mathbf{run}(\langle \alpha{:}\overline{*} \vdash \mathbf{fst}(1, \lambda x{:}\alpha\ .\ x) : \mathbf{Nat}\rangle, 0 : \mathbf{Nat})$$

Naïvely rewriting this to:

$$\mathbf{fst}(1, \lambda x{:}\alpha\ .\ x)$$

would introduce a free type variable $\alpha$. To preserve subject reduction we cannot allow such residual free type variables to escape the scope of the deferred expression. However, since $t_0$ must be parametric on any such free type variables we may safely instantiate each of them to $\mathbf{0}$, the empty type. Thus rule run actually rewrites the above example as:

$$\mathbf{fst}(1, \lambda x{:}\mathbf{0}\ .\ \mathbf{x})$$

Rule run/fail catches an attempt to run an ill-typed deferred expression, and evaluates the exception expression immediately.

#### 5.4.3 Reduction

Relation $\longrightarrow$ induces the *one step rewrite relation*, $\longmapsto$, defined as

$$E_0[t] \longmapsto E_0[u] \text{ iff } t \longrightarrow u$$

Notice that context $E_0$ allows rewrites in any term at stage 0, even if deeply nested within matching $\langle\rangle$ brackets and $\sim$ splices. We write $\mapsto\!\!\!\!\rightarrow$ for the reflexive, transitive closure of $\longmapsto$

There is a slight thorn in our side here. We would like $\mapsto\!\!\!\!\rightarrow$ to induce an evaluation *function*

$$\mathit{eval}(t) = v \text{ iff } t \mapsto\!\!\!\!\rightarrow v$$

though unfortunately $E_0$ is not deterministic in the location of the hole within it. For example:

$$\langle(\sim f)(\sim x)\rangle$$

has two possible decompositions:

$$\langle E_1[\sim f]\rangle \quad \text{and} \quad \langle E_1[\sim x]\rangle$$

This is easily fixed (see [25]) by defining a family of terms and contexts, indexed by stage number. In this paper however we choose to ignore the problem, and assume that when faced with a choice between splice redexes, *eval* chooses the leftmost.

### 5.5 Syntactic Soundness

We now show our type system and operational semantics are in agreement. The full proof is in [25].

**Lemma 1 ($\longrightarrow$ Type Preserving)** If $\Gamma \vdash_n t : \sigma$ and $t \longrightarrow u$ then $\Gamma \vdash_n u : \sigma$

**Proof** By case analysis on $t$. Most cases are standard, and make use of the usual lemmas for term and type substitution, extended to handle our multi-stage system.

The proof for splice requires additional lemmas dealing with contexts, type unification, and the key lemma that in $\langle \overline{\alpha{:}*} \vdash E_1[t] : \tau\rangle$, $E_1$ cannot introduce type variables into the scope of $t$. This allows us to show $rng(\Theta) \subseteq \overline{\beta} \cup ftv(\Gamma)$, and thus the rewrite does not allow type variables to escape the scope of the outer deferred term.

The proof for run requires the small lemma that if $\cdot \vdash_1 t : \tau$ then $\cdot \vdash_0 t : \tau$.

**Lemma 2 ($\longrightarrow$ Progress)** If $\cdot \vdash_0 t_0 : \sigma$ then either $t_0$ is a value or there exists a unique context $E_0$ and terms $u_0$ and $w_0$ s.t. $t_0 = E_0[u_0]$ and $u_0 \longrightarrow w_0$.

**Proof** By induction on $t_0$. Most cases are straightforward. A lemma showing that if a term is splice-full then there exists a context $E_1$ reaching the leftmost splice expression within it is required when $t_0$ is a deferred expression.

**Theorem 3 (Syntactic Soundness)** If $\cdot \vdash_0 t : \sigma$ then either $t$ diverges or $t \mapsto\!\!\!\!\to v$ and $\cdot \vdash_0 v : \sigma$.

**Theorem 4 (Subject Reduction)** If $\Gamma \vdash_n t : \sigma$ and $t \mapsto\!\!\!\!\to u$ then $\Gamma \vdash_n u : \sigma$.

We should also ensure $\lambda$dyn remains faithful to the naïve semantics of §2.3 despite the optimisations we have made. In particular, run should succeed on exactly the same terms as before. Unfortunately this would also require a rigorous development of the naïve semantics, and we defer this to future work.

## 6 Subtleties

In this section we address some of the more subtle aspects of type checking we encountered in §5.

### 6.1 Dynamic Let-Generalisation

In §5.1 we were careful to prevent spice-full terms being **let** bound. We now motivate this restriction.

Consider the following (illegal) term:

$$\langle \mathbf{let}\, f = \lambda x\,.\, \sim g\ x\, \mathbf{in}(f\ 1, f\ \mathbf{true}) : \tau\rangle$$

and consider how $\tau$ depends on $g$:

| $g$ | $\tau$ |
|---|---|
| $\langle \lambda y\,.\,y\rangle$ | $(\mathbf{Nat}, \mathbf{Bool})$ |
| $\langle \lambda y\,.\,1\rangle$ | $(\mathbf{Nat}, \mathbf{Nat})$ |
| $\langle \lambda y\,.\,y+1\rangle$ | ill-typed |

Clearly $\tau$ depends both on the type of the body of $g$, and the types at which $f$ is called. How can this dependency be captured?

One solution is to type generalise $f$ just before it is to be run. However, this takes us back to our naïve semantics of §2.3, with the associated drawbacks.

Another solution is express this dependency by a functor constructed by higher-order unification during splicing. If we let $F$ be a free functor variable, our example could be annotated as:

$$\begin{array}{l}\langle \mathbf{let}\, f{:}\forall\alpha\,.\,\alpha \to F(\alpha) = \lambda x{:}\alpha(\sim g{:}\alpha \to F(\alpha))\ x \\ \quad \mathbf{in}(f\ \mathbf{Nat}\ 1, f\ \mathbf{Bool}\ \mathbf{true}) : (F(\mathbf{Nat}), F(\mathbf{Bool}))\rangle\end{array}$$

Now the dependency between $g$ and the types $f$ is specialised to has been made explicit. Should $g$ evaluate to $\langle \lambda y\,.\,1\rangle$, splicing will construct the higher-order unification problem

$$\beta \to \mathbf{Nat} = \alpha \to F(\alpha)$$

which has the unique solution

$$[\beta \mapsto \alpha, F \mapsto \lambda\gamma{:}*\,.\,\mathbf{Nat}]$$

resulting in the same type as before.

But if $g$ evaluates to $\langle \lambda y\,.\,y+1\rangle$ then the problem becomes

$$\mathbf{Nat} \to \mathbf{Nat} = \alpha \to F(\alpha)$$

which now has the two solutions

$$\begin{array}{l}[\alpha \mapsto \mathbf{Nat}, F \mapsto \lambda\gamma{:}*\,.\,\gamma] \\ [\alpha \mapsto \mathbf{Nat}, F \mapsto \lambda\gamma{:}*\,.\,\mathbf{Nat}]\end{array}$$

As $F$ may appear within an arbitrary number of splice expressions, the complexity of constraint satisfaction is at least exponential. Abadi *et al* [2] introduced a sufficient syntactic condition on the use of functor variables to guarantee the *definiteness* of solutions to these unification problems.

We argue that our approach, to reject such terms altogether, simplifies our semantics greatly, allows significant implementation optimisations, and does not reduce expressibility critically. In particular, since defer expressions tend to be small, let-bound splice-full terms may be unfolded into their bodies without risk of excessive code duplication.

Importantly, this restriction only applies to let-bound terms within a defer expression, and *not* to the way defer expressions are spliced or run. Compare this with the system of Abadi *et al* [2], which *requires* higher-order unification to be able to manage dynamic values of polymorphic type.

In practice this restriction is too severe; indeed the careful reader will have noticed the **let** bound expressions in the *spectrograph* example of §3.2 are not splice-free. A practical implementation need only prevent type generalising **let** bound splice-full expressions. Both monomorphic splice-full expressions, and polymorphic splice-free terms may be safely **let** bound.

### 6.2 Closed vs Open Code

In §4 we argued for incremental staged type inference as a replacement for the naïve semantics of §2.3. Looking again at the naïve rewrite rule for run we see $t_0'$ is type checked within the empty environment, implicitly requiring $t_0'$ to be closed. When this was replaced with the rewrite rule run given in Figure 7, we see that this implicit test has become explicit. However, the explicit test could be expensive, perhaps even $O(|t_0|)$. Unfortunately, eliding this test in the incremental setting destroys our subject-reduction property.

For example, evaluating

$$\langle \lambda x\ .\ \text{\textasciitilde}\,\mathbf{run}(\langle x\rangle, \langle 0\rangle)\rangle \qquad \textit{well-typed}$$

will first evaluate $\mathbf{run}(\langle x\rangle, \langle 0\rangle)$. If this were to succeed and yield $x$, the result would be binding-time incorrect:

$$\langle \lambda x\ .\ \text{\textasciitilde} x\rangle \qquad \textit{ill-typed}$$

Here $x$ is bound at stage 1, but used at stage 0. The problem is that $\langle x\rangle$ depends on an environment binding $x$, which is lost by the **run**.

Thus, the body of a deferred expression which is to be **run** must not depend on any variables bound at its stage or later, as these bindings will be lost. It is safe to depend on variables bound at earlier stages as, thanks to our call-by-value semantics, these variables will have been replaced with values by the time the defer expression is evaluated.

The above example could be rejected syntactically, as $\langle x\rangle$ is obviously open. But expressions such as:

$$\mathbf{let}\, f\ \mathit{code} = \mathbf{run}(\mathit{code}, \langle 0\rangle)\,\mathbf{in}\langle \lambda x\ .\ \text{\textasciitilde}(f\ \langle x\rangle)\rangle$$

clearly require the constraint that *code* be closed to be propagated to $f$'s call-site, suggesting this check belongs in the type system.

We briefly outline an enhancement to the type system of §5.3 which will reject examples such as these at compile-time instead of run-time. This is based on a type system developed in previous work [26].

The type $\langle\rangle$ of deferred expressions is now partitioned into a type $\langle\epsilon\rangle$ for each *environment* $\epsilon$. The type checker keeps track of the environment each variable belongs to, in addition to its usual type and stage. Type annotation will assign a deferred expression $\langle t\rangle$ the type $\langle\epsilon\rangle$ if $t$ uses any variable from environment $\epsilon$. Importantly, if $t$ uses only locally bound variables, then $\epsilon$ will not appear in its type context, *i.e.*, $\epsilon$ will be free.

Thus to accept only closed deferred expressions, the type rule for **run** simply checks its argument has type $\langle\epsilon\rangle$ for $\epsilon$ free. That is, **run** has the rank-2 polymorphic type:

$$\mathbf{run} : \forall\alpha\ .\ (\forall\epsilon\ .\ \langle\epsilon\rangle, \alpha) \to \alpha$$

confirming our intuition that a runnable dynamic expression must be runnable *in all* environments.

A formal description of this type system can be found in [25].

Unfortunately, in the dynamic context, the rank-2 polymorphism of **run** very quickly spreads to other functions, requiring their type to be given as an explicit annotation, and requiring a very much more sophisticated type system to deal with them [23]. This clearly contradicts our claim that we require "no explicit types". For this reason we prefer to leave the check for closed code within the run rewrite rule.

## 7 Related Work

Much work in dynamic typing has concentrated on *soft typing*: static analyses which determine when run-time type checks may be safely elided from programs in untyped languages [8, 4, 14].

Solutions to the dual problem of adding dynamic types to a statically typed language appear in CLU and Cedar/Mesa (see [1] for a discussion), and were first considered for ML in unpublished work of Mycroft [21]. Most proposals to date have been variations on the **typecase** construct illustrated in the introduction. These vary in their "resolving power": the ability to distinguish dynamic values based on their type within a **typecase**.

At the simpler end of the spectrum is the system of Abadi *et al* [1], which allows **typecase** patterns to contain type variables in order to express type dependencies within and between dynamic values. This means a family of dynamic values may be matched by a single **typecase** arm. However all dynamic values must be monomorphic.

Two approaches to supporting polymorphic dynamic values have been explored by Leroy and Mauny [17]. Their simplest system allows a polymorphic dynamic value to be distinguished from all of its instances, and also allows a polymorphic dynamic value to be type specialised when pattern matching. Their more complex system also allows type variables within patterns much as in [1]. Pattern matching within **typecase** is formalised as first-order unification under a mixed quantifier prefix. Universally quantified variables denote where a pattern may match a family of dynamic values, where as existentially quantified variables denote where a pattern may match only suitably polymorphic dynamic values.

Dubois *et al* [11] allow functions to implicitly bind the types of their arguments, and also provide a **typecase** construct to support generic programming.

The extension of [1] to arbitrary polymorphic dynamic values has also been considered by Abadi *et al* [2]. Their proposal extends patterns with higher order functor variables in order to match dynamic values with arbitrary polymorphic types. This adds a significant complexity to the language, and functors must be syntactically restricted to avoid requiring full higher-order unification. For example:

$$\begin{array}{l}
\mathit{apply} : \mathbf{Dyn} \to \mathbf{Dyn} \to \mathbf{Dyn} \\
\mathit{apply}\ \mathit{df}\ \mathit{dx} =\ \mathbf{typecase}\, \mathit{df}\, \mathbf{of}\{ \\
\quad \{F, G\}(f{:}\forall Z\ .\ F(Z) \to G(Z)) \to \\
\qquad \mathbf{typecase}\, \mathit{dx}\, \mathbf{of}\{ \\
\qquad\quad \{T\}(x{:}T) \to\ \mathbf{dynamic}(f\ [T]\ x){:}G(T)\}\}
\end{array}$$

applies a dynamic value to a dynamic function polymorphic on a single type variable, encoding the required type check within the **typecase** patterns. $F$ and $G$ are bound to functors at run-time, $f$ is explicitly type specialised to $T$, and the type of the result is explicitly recorded in the resulting dynamic value. This approach requires the programmer to perform complex type checking explicitly. It is also fairly verbose compared with our solution:

$$\begin{array}{l}
\mathit{apply} : \langle\rangle \to \langle\rangle \to \langle\rangle \\
\mathit{apply}\ \mathit{df}\ \mathit{dx} = \langle \text{\textasciitilde}\mathit{df}\ \text{\textasciitilde}\mathit{dx}\rangle
\end{array}$$

Our approach is motivated by the recent work of Davies and Pfenning [10, 9] and Taha and Sheard [26] in staged computation. Our $\langle\rangle$ and ~ operators correspond with the **next** and **prev** constructs of [10], extended to allow the *cross-stage persistence* of [26]. Our **run** operator corresponds to the **unbox pop** operation of [10], with an appropriate dynamic check for closure.

## 8 Conclusions and Future Work

We presented an approach to dynamic typing which extends staged computation to also stage type inference, and demonstrated its convenience in programming examples for which type information is difficult (*e.g.*, *sprintf* §3.1) or impossible (*e.g.*, *getCode* §3.2) to express at compile-time. We also demonstrated how the additional expressiveness of staged computation (splicing, open code, and multiple stages) can make programming with dynamically typed values even more convenient.

Our system definitely lies at the more complex end of the spectrum of dynamic typing systems sketched in §7. Is this complexity justified? If only monomorphic dynamic values are required, and these are simply to be stored, retrieved or printed, then the answer is probably no. Once polymorphism is introduced the picture is less clear. We argue that our proposal is conceptually much simpler than that of Abadi *et al* [2]. We also argue that the additional expressiveness of our system over that of Leroy and Mauny [17] can be exploited to add features such as pretty-printing (§3.1) and dynamic code linking (§3.2) to a language simply by the addition of a few primitive functions.

There are three main avenues for future research. We currently only have an interpreted implementation of a superset of $\lambda$dyn, so its feasibility will only be known once an abstract machine and compiled implementation exist. Implementing code splicing and run-time type unification efficiently are the primary challenges. The work of Lee and Leone on run-time code generation [16] suggests the overhead for splicing may be only a small factor, and the extensive body of research on efficient unification for Prolog suggests type unification could also be inexpensive.

We wish to develop a denotational semantics and semantic soundness theorem to complement the purely syntactical presentation of this paper. Models for monomorphic dynamic values [1] and two-level languages [20] have been developed in isolation. Their combination and generalisation to polymorphic dynamic values is non-trivial. Note, however, that their generalisation to arbitrary stages instead of just two should not be problematic, as all stages $> 0$ may be collapsed to a single dynamic stage.

Finally, we have presented $\lambda$dyn as a call-by-value calculus. Using the well known embedding of the call-by-name lambda-calculus into the call-by-value calculus [13], we could use $\lambda$dyn as an intermediate language for a call-by-name language in which let-bound expressions (which create thunks or closures, which we represent as deferred expressions) could be tagged as statically or dynamically typed. This could be used to add a simple form of dynamic typing to Haskell.

## Acknowledgements

We thank Walid Taha and the anonymous referees for their helpful comments on this work.

## References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Apr 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, Jan 1995.

[3] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[4] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, Oregon*, pages 163–173, 1994.

[5] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[6] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford Science Publishers, 1992.

[7] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1/2):4–56, 1994.

[8] R. Cartwright and M. Fagan. Soft typing. In *ACM SIGPLAN-91 Conference on Programming Language Design and Implementation, Toronto, Ontario*, pages 278–292. ACM Press, Jun 1991.

[9] R. Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey*, pages 184–195. IEEE Computer Society Press, Jul 1996.

[10] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 258–270. ACM Press, Jan 1996.

[11] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 118–129. ACM Press, Jan 1995.

[12] J.-Y. Girard. The system F of variable types, Fifteen years later. In G. Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, chapter 6. Addison-Wesley, 1990.

[13] J. Hatcliff and O. Danvy. Thunks and the lambda-calculus. *Journal of Functional Programming*, 7(3):303–319, May 1997.

$$\frac{m \le n \quad \overline{\beta} \cap \mathit{ftv}(\Gamma) = \emptyset}{\mathbf{Id} \mid \Gamma, x : (\forall \overline{\alpha} \,.\, \tau, m) \vdash_n x \hookrightarrow x\, \overline{\beta} : \tau[\overline{\alpha} \mapsto \overline{\beta}]}\ \textsf{var}$$

$$\frac{\begin{array}{c}\Theta_1 \mid \Gamma \vdash_n e \hookrightarrow t : \tau \quad \mathit{spliceFree}(t) \\ \overline{\alpha} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Theta_1\Gamma) \quad \sigma = \forall \overline{\alpha} \,.\, \tau \\ \Theta_2 \mid \Gamma, x : (\sigma, n) \vdash_n f \hookrightarrow u : \upsilon \quad \Theta = \Theta_1 \sqcup \Theta_2\end{array}}{\Theta \mid \Gamma \vdash_n \mathbf{let}\, x = e \,\mathbf{in}\, f \hookrightarrow \mathbf{let}\, x{:}\sigma = t \,\mathbf{in}\, u : \Theta\upsilon}\ \textsf{let}$$

$$\frac{\Theta \mid \Gamma \vdash_{n+1} e \hookrightarrow t : \tau \quad \overline{\alpha} = \mathit{ftv}(t, \tau) \setminus \mathit{ftv}(\Theta\Gamma)}{\Theta \mid \Gamma \vdash_n \langle e \rangle \hookrightarrow \langle \overline{\alpha}{:}\overline{*} \vdash t : \tau \rangle : \langle\rangle}\ \textsf{defer}$$

$$\frac{\begin{array}{c}\alpha \notin \mathit{ftv}(\Gamma) \quad \Theta_1 \mid \Gamma \vdash_n e \hookrightarrow t : \tau \\ \Theta_2 = \mathit{unify}(\tau, \langle\rangle) \quad \Theta = \Theta_1 \sqcup \Theta_2\end{array}}{\Theta \mid \Gamma \vdash_{n+1} \mathord{\sim} e \hookrightarrow (\mathord{\sim} t{:}\alpha) : \alpha}\ \textsf{splice}$$

$$\frac{\begin{array}{c}\Theta_1 \mid \Gamma \vdash_n e \hookrightarrow t : \upsilon \quad \Theta_2 = \mathit{unify}(\upsilon, \langle\rangle) \\ \Theta_3 \mid \Gamma \vdash_n f \hookrightarrow u : \tau \quad \Theta = \Theta_1 \sqcup \Theta_2 \sqcup \Theta_3\end{array}}{\Theta \mid \Gamma \vdash_n \mathbf{run}(e, f) \hookrightarrow \mathbf{run}(t, u{:}\Theta\tau) : \Theta\tau}\ \textsf{run}$$

Figure 8: Type annotation of λdyn terms (extract)


[14] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture*, pages 192–203. ACM Press, Jun 1995.

[15] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[16] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *ACM SIGPLAN-96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, Pennsylvania, May 1996.

[17] X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[18] Microsoft. The component object model specification. Technical report, Microsoft Corporation, 1995.

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[20] E. Moggi. A categorical account of two-level languages. In *Proceedings of the Thirteenth Annual Conference on Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science Volume 6. Elsevier Science Publishers, 1997.

[21] A. Mycroft. Dynamic types in ML. (unpublished draft article), 1983.

[22] F. Nielson and H. R. Nielson. *Two-level Functional Languages.* Cambridge University Press, 1992.

[23] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 54–67. ACM Press, Jan 1996.

[24] J. Peterson and K. Hammond. *Report on the Programming Language Haskell (Version 1.4)*, Apr 1997.

[25] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. Technical Report TR-1997-26, University of Glasgow, Department of Computing Science, Aug 1997. Available from `http://www.dcs.gla.ac.uk/~mbs/pub/tr_97_26.ps.gz`.

[26] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics Based Program Analysis*, pages 203–217. ACM Press, 1997.

[27] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.


## A Type Annotation

Figure 8 presents our type reconstruction algorithm, a straightforward extension to Algorithm W [19]. We use the deductive style, motivated by the presentation in [23]. We elide rules const, abs, app and letrec as their definition is straightforward.

We intend the judgement $\Theta \mid \Gamma \vdash_n e \hookrightarrow t : \tau$ to be true when source term $e$ is rewritten to annotated term $t$ with *most general* type $\tau$ in a type context $\Theta\Gamma$. $\Lambda\overline{\alpha}{:}\overline{*} \,.\, t$ should then be well typed under $\overline{\alpha}{:}(n, \overline{*}), \Theta\Gamma$ by the rules of Figure 5, where $\overline{\alpha} = \mathit{ftv}(t, \tau)$. A quick check shows the relation to be functional, with $\Gamma$ and $e$ as inputs, and $\Theta$, $t$ and $\tau$ as outputs.

Rules splice and run annotate their terms by the most general type inferred from their context. Rule defer annotates a deferred expression by the most general type of its body, and collects all type variables in its type or body not bound in an outer scope. This step mimics the generalisation done by let on let-bound terms.

Rule defer will reject as ill-typed any defer expression whose body is statically ill-typed. For example $\langle \lambda x \,.\, 1\; x \rangle$ is rejected rather than annotated as the well-typed but probably useless term $\langle \mathbf{Fail} \rangle$.

Our system enjoys a principle type property, but we defer showing soundness and completeness of type annotation w.r.t. type checking to further work.