Refined Environment Classifiers

Type- and Scope-safe Code Generation with Mutable Cells

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo

[1] Tohoku University oleg@okmij.org
[2] University of Tsukuba kameyama@acm.org

Abstract. Generating high-performance code and applying typical optimizations within the bodies of loops and functions involves moving or storing open code for later use, often in a different binding environment. There are ample opportunities for variables being left unbound or accidentally captured. It has been a tough challenge to statically ensure that by construction the generated code is nevertheless well-typed and *well-scoped*: all free variables in manipulated and stored code fragments shall eventually be bound, by their intended binders.
We present the calculus for code generation with mutable state that for the first time achieves type-safety and hygiene without ad hoc restrictions. The calculus strongly resembles region-based memory management, but with the orders of magnitude simpler proofs. It employs the rightly abstract representation for free variables, which, like hypothesis in natural deduction, are free from the bureaucracy of syntax imposed by the type environment or numbering conventions.
Although the calculus was designed for the sake of formalization and is deliberately bare-bone, it turns out easily implementable and not too bothersome for writing realistic program.

1 Introduction

Code generation exhibits the all-too-common trade-off: obtaining code with the highest performance; statically ensuring the code quality; being able to use the code-generating system in practice – choose two. Optimizing compilers and many practical code-generating tools do all desired optimizations. The correctness of the result is ensured however only by careful programming. This is not a problem in case of a compiler written by a small team of experts and changed relatively infrequently. The lack of static assurances is worrisome for code transformation and generation libraries written by domain experts, who have less time to devote to proofs and have to continually tune their libraries to the domain knowledge and circumstances. There is the attested danger of generating code with unbound, or worse, unexpectedly bound variables. At the very least, we would like to guarantee that the generated code – at all times, even the code fragments – is well-formed, well-typed, and all of its free variables will eventually be bound by their intended binders. This guarantee should hold before we compile the generated code, which is typically unfit for human reading. Ideally, the guarantee should hold even before we compile the generator.

On the other side of the trade-off are the staged calculi such as λ° and λ^α [4, 15] that express code generators with the desired static guarantees. They are called 'staged' because evaluation is stratified: the result of the present, or Level-0, stage is the code to be evaluated at the next, Level-1, or future stage. The calculi have been implemented as full-featured staged languages used in practice [8, 17]. Another example is Pouillard's [13] code generation and analysis library with proven correctness. Alas, all these systems restrict the range of safe operations on open code: in particular, they limit or outlaw the operations that move or store open code, retrieving it later in a different binding environment. Such operations are required for many optimizations such as let-insertion, memoization, loop interchange and tiling. There have been general approaches that permit the desired open code motions and provide static guarantees: for example, [12]. Alas, they are too complex to use in practice or even to implement. For more discussion, see §5 and especially [6].

StagedHaskell [6] overcomes the impasse, but partially. It is the library for code generation that supports code movements, including movements via any computational (monadic) effect. Using a contextual modal type system, the library statically assures that at all times the generated code is well-formed, well-typed and *well-scoped*: all free variables in manipulated and stored code fragments shall eventually be bound by their intended binders. However, the safety properties have been argued only informally. The main reason is that the complexity of the Haskell implementation, specifically, the encoding of the contextual modal type system, make the formal reasoning difficult.

The present paper takes the first step of formalizing StagedHaskell: it distills the staged calculus `<NJ>` that safely permits open code movements across different binding environments via mutable cells. The calculus can express realistic examples from StagedHaskell, such as the assert-insertion, see §4.

Although `<NJ>` was motivated by code generation with safety guarantees, it turned a vantage point to view seemingly unrelated areas. First, there is an uncanny similarity between generating code of functions (or other blocks with local binders) and region-based memory management. Preventing the 'extrusion' of free variables out of the bodies of generated functions is similar to keeping reference cells allocated within a region from leaking out. We have consciously used this similarity, adapting techniques from region calculi [5].

The key to ensuring hygiene and type safety when manipulating open code is reflecting free variables of a code fragment in its type – which evokes contextual modal type theory [10] and, in general, sequent calculus. The structural rules such as weakening now turn up in programs, e.g., as 'shifts' of De Bruijn indices [3]. After all, in metaprogramming, meta-level becomes the object level. 'The bureaucracy of syntax' now worries not only logicians but also programmers.

A particularly elegant method to overcome the complexities and redundancies of concrete name and environment representations is environment classifiers [15] (recalled and discussed in §3.2). A single classifier represents a set of free variables, abstracting from their order, quantity, or names. Unfortunately, in the presence of effects, the original environment classifiers are too coarse, abstracting

away the information needed to ensure the type safety of effectful generators. Inspired by the concept of local assumptions from Natural Deduction NJ, we have identified the minimal necessary refinement of environment classifiers.

Contributions Our specific contributions are as follows:

- Practical two-stage calculus `<NJ>` whose type system statically ensures hygiene and the type-safety of the generated code in the presence of mutable reference cells. The calculus distills the design of the practical StagedHaskell library. The calculus itself is easily implementable.
- Refinement of environment classifiers – imposing partial order – that preserves all their simplicity and advantages and is compatible with effects.

`<NJ>` is close to the current MetaOCaml [8], which permits leaking of variables (scope extrusion) but raises a run-time error at the moment the code with leaked (extruded) variables is about to be used. Our calculus prevents such errors statically.

The calculus has been implemented as a simple embedding in OCaml, whose type checker checks `<NJ>` types and even infers them. Signatures are only needed for functions that receive code values as arguments *and* use them in distinct binding environments. One is immediately reminded of ML^F [9]; this is not an accident, as we shall see in §3.3. All examples in the paper are slightly reformatted running code. The implementation, with more examples, is available at `http://okmij.org/ftp/tagless-final/TaglessStaged/metaNJ.ml`.

This paper is organized as follows: The next section introduces the calculus, using many examples to illustrate its syntax and dynamic semantics. §3 describes the type systems and proves its soundness. §3.1 specifically demonstrates the obvious and very subtle dangers arising from storing open code in mutable cells, and how `<NJ>` prevents the dangers but not the free use of reference cells. Responsible for this are refined environment classifiers; §3.2 discusses what, why and how. §4 shows off a complex example: It is used exactly as was explained in [6], deliberately to demonstrate that `<NJ>` is capable of representing practical StagedHaskell examples.

2 `<NJ>`, its Syntax and Semantics

Formally, the syntax of `<NJ>` is defined in Fig. 1. This section introduces the calculus and its dynamic semantics more accessibly, on a series of small examples.

`<NJ>` is a lambda-calculus with reference cells and special constants to create and combine code values. Whereas $1 : \mathsf{int}$ is the familiar constant of the base type int, $\underline{\mathsf{cint}}\ 1$ is an expression of the code type $\langle \mathsf{int} \rangle^{\gamma}$ which evaluates to $\langle 1 \rangle$. The code types are explained in §3. Here, $\underline{\mathsf{cint}}$ is a special code-generating constant, also called code combinator[3] [16, 18]. We underline all such constants. Likewise, $\underline{\mathsf{cbool}}\ \mathbf{true}$ evaluates to $\langle \mathbf{true} \rangle$. Since the $\underline{\mathsf{cint}}$ and $\underline{\mathsf{cbool}}$ expressions are common

[3] The StagedHaskell library, the prototype of `<NJ>`, is a code-combinator library.

we adopt the abbreviated notation % that stands for either constant, depending on the context.

The bracketed expressions like $\langle 1 \rangle$ cannot appear in source programs; they come only during and as the result of reductions. This is the most visible distinction from stage calculi like λ^α [15] and MetaOCaml. Neither do we have 'splicing' (or, 'escape', unquotation). Bracketed expressions are essentially constants: they cannot be decomposed, inspected, or substituted into. Only a subset of expressions may be bracketed: underlined constants and the brackets themselves are excluded. <NJ> therefore is the two-stage calculus, for generating code but not for generating code generators[4]. We will sometimes use the superscript e^1 to emphasize the distinction between the host language and the generated language. Most of the time the superscript is elided for ease of notation.

Figure 2 defines the constants of <NJ>. They come in different arities. Seen earlier 1 and **true** are zero-ary constants, denoted as c_0 in Fig. 1. On the other hand, $\underline{\mathsf{cint}}$ and $\underline{\mathsf{cbool}}$ have the arity 1, and are not considered expressions per se: only their applications to one argument are expressions, see Fig. 1. Likewise, + is an arity-2 constant, requiring two arguments to be regarded as an expression. We write such an expression in the conventional infix notation 1+2, which evaluates to 3. Besides $\underline{\mathsf{cint}}$ and $\underline{\mathsf{cbool}}$, there are constants that combine already built code values: $\%1 \mathbin{\underline{+}} \%2$ of the code type $\langle \mathsf{int} \rangle^\gamma$ evaluates as follows according to the rules of Fig. 3 and 4.

$$\%1 \mathbin{\underline{+}} \%2 \rightsquigarrow \langle 1 \rangle \mathbin{\underline{+}} \%2 \rightsquigarrow \langle 1 \rangle \mathbin{\underline{+}} \langle 2 \rangle \rightsquigarrow \langle 1{+}2 \rangle$$

Here, $\underline{+}$ is an arity-2 constant, which we also write in infix. Again, all constants must be fully applied: there are no partial applications, sections, or other sugar.

<NJ> is the lambda-calculus, with the standard abstractions $\lambda \mathsf{x}.\ \mathsf{e}$ and applications $\mathsf{e}_1\ \mathsf{e}_2$. We let **let** $\mathsf{x} = \mathsf{e}_1$ **in** e_2 stand as the abbreviation for $(\lambda \mathsf{x}.\ \mathsf{e}_2)\ \mathsf{e}_1$, and $\mathsf{e}_1; \mathsf{e}_2$ for **let** $\mathsf{x} = \mathsf{e}_1$ **in** e_2 where x does not appear free in e_2. The semantics of <NJ> is the standard small-step left-to-right call-by-value, see Fig. 3. (Heaps will be explained later on and can be ignored for now). <NJ> can also generate the code of functions, using the expression form $\underline{\lambda} \mathsf{x}.\ \mathsf{e}$ with peculiar semantics, which we explain in detail. For example, the expression $\underline{\lambda} \mathsf{x}.\ \mathsf{x} \mathbin{\underline{+}} \%3$ eventually generates the code of the function that increments its argument by 3:

$$\underline{\lambda} \mathsf{x}.\ \mathsf{x} \mathbin{\underline{+}} \%3 \rightsquigarrow \underline{\lambda} \mathsf{y}.\langle \mathsf{y} \rangle \mathbin{\underline{+}} \%3 \rightsquigarrow \underline{\lambda} \mathsf{y}.\langle \mathsf{y} \rangle \mathbin{\underline{+}} \langle 3 \rangle \rightsquigarrow \underline{\lambda} \mathsf{y}.\langle \mathsf{y}{+}3 \rangle \rightsquigarrow \langle \lambda \mathsf{y}.\ \mathsf{y}{+}3 \rangle$$

First, the expression $\underline{\lambda} \mathsf{x}.\ \mathsf{x} \mathbin{\underline{+}} \%3$ reduces by choosing a fresh variable name y, replacing all free occurrences of x in its body with $\langle \mathsf{y} \rangle$ and wrapping the result in $\underline{\lambda} \mathsf{y}$. (Expressions of the form $\underline{\lambda} \mathsf{y}.\mathsf{e}$ come up during the evaluation and do not appear in source programs.) Next, the body of thus built $\underline{\lambda} \mathsf{y}.\langle \mathsf{y} \rangle \mathbin{\underline{+}} \%3$ reduces as described earlier. The final reduction in the sequence builds the resulting code. Thus producing the code for functions has two separate phases: generating the name for the bound variable, and generating the binder for that name at the

[4] This restriction certainly simplifies the formalism. It is also realistic: in all our experience of using MetaOCaml, the multi-stage language, we are yet to come across any real-life example needing more than two stages. Template Haskell is also two-stage.

end. In many staged calculi the two phases can be (and are) combined. The effects force them apart however, as we shall see soon.

Variables	x,y,z,u,f,n,r...
Classifier	γ
Location	l
Types	$t ::= \mathsf{int} \mid \mathsf{bool} \mid t \to t \mid t\ \mathbf{ref} \mid \langle t \rangle^\gamma$
Level 1 Types	$t^1 ::= \mathsf{int} \mid \mathsf{bool} \mid t^1 \to t^1 \mid t^1\ \mathbf{ref}$
Level 0 Expressions	$e ::= x \mid l \mid c_0 \mid c_1\ e \mid c_2\ e\ e \mid c_3\ e\ e\ e \mid \lambda x.\ e \mid \underline{\lambda} x.\ e \mid e\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$
Internal Expressions	$\underline{\underline{\lambda}} x.\ e \mid \langle e^1 \rangle$
Level 1 Expressions	$e^1 ::=$ e without code combinators, internal expressions, locations
Values	$v ::= c_0 \mid l \mid \lambda x.\ e \mid \langle e \rangle$

Fig. 1. Syntax of <NJ>. The constants c_i with their arities i are defined in Fig. 2.

Arity 0	Arity 2
1,2,3,... : int	$+$: $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$
true, **false**: bool	$\underline{+}$: $\langle \mathsf{int} \rangle^\gamma \to \langle \mathsf{int} \rangle^\gamma \to \langle \mathsf{int} \rangle^\gamma$
Arity 1	$=$: $\mathsf{int} \to \mathsf{int} \to \mathsf{bool}$
$\underline{\mathsf{cint}}$: $\mathsf{int} \to \langle \mathsf{int} \rangle^\gamma$	$\underline{=}$: $\langle \mathsf{int} \rangle^\gamma \to \langle \mathsf{int} \rangle^\gamma \to \langle \mathsf{bool} \rangle^\gamma$
$\underline{\mathsf{cbool}}$: $\mathsf{bool} \to \langle \mathsf{bool} \rangle^\gamma$	$\underline{@}$: $\langle t_1 \to t_2 \rangle^\gamma \to \langle t_1 \rangle^\gamma \to \langle t_2 \rangle^\gamma$
fix: $((t_1 \to t_2) \to (t_1 \to t_2)) \to (t_1 \to t_2)$	:=: $t\ \mathbf{ref} \to t \to t$
$\underline{\mathsf{fix}}$: $\langle (t_1 \to t_2) \to (t_1 \to t_2) \rangle^\gamma \to \langle t_1 \to t_2 \rangle^\gamma$	$\underline{:=}$: $\langle t\ \mathbf{ref} \rangle^\gamma \to \langle t \rangle^\gamma \to \langle t \rangle^\gamma$
$\underline{!}$: $\langle t\ \mathbf{ref} \rangle^\gamma \to \langle t \rangle^\gamma$	
ref: $t \to t\ \mathbf{ref}$	Arity 3
$\underline{\mathbf{ref}}$: $\langle t \rangle^\gamma \to \langle t\ \mathbf{ref} \rangle^\gamma$	$\underline{\mathsf{cif}}$: $\langle \mathsf{bool} \rangle^\gamma \to \langle t \rangle^\gamma \to \langle t \rangle^\gamma \to \langle t \rangle^\gamma$

Fig. 2. The constants c_i of <NJ> with their arities i. The underlined constants, whose result type is code type, are code combinators. The shown types are schematic: t denotes any suitable type and γ any suitable classifier. We silently add other arithmetic and comparison constants and code combinators, similar to + and =. Although the constants may have function types, they are not expressions, unless applied to the right number of arguments.

For another illustration we take the familiar power example: generating a function that raises its argument to the given power by repeated multiplications.

```
let body = λf n x. if n=0 then %1 else x *_ f (n−1) x in
λn. λ_x. (fix body) n x
```

Applying the result to, say, 3 produces $\langle \lambda y.\ y * y * y * 1 \rangle$.

<NJ> has mutable state in the form of the familiar mutable cells, such as those found in ML and many other languages. Correspondingly, the calculus has the form **ref** e to create a fresh reference cell holding the value of e, ! e to dereference it, obtaining the held value, and $e_1 := e_2$ to replace the value of the cell e_1 with the value of e_2, returning the latter value. The semantics is standard, involving locations l and the heap H, the finite map from locations to values. The empty heap is denoted as []; (l:v,H) is the heap that contains the association of l with v plus the associations in H. The domain of the latter does not include l. From λ^U [2] we borrow the heap-like *name heap* N, which is the set of names used for variables in the generated code. As we shall see throughout the paper, there is an uncanny similarity between reference cells and the future-stage variable names.

Contexts
$$E ::= [] \mid E\ e \mid v\ E \mid c_1\ E \mid c_2\ E\ e \mid c_2\ v\ E \mid \textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e \mid \underline{\lambda}x.\ E$$

Heap
$$H ::= [] \mid l{:}v,H$$

Names
$$N ::= [] \mid y,N$$

Substitutions
$$e_1[x{:=}e_2]$$

Reductions
$$N_1;H_1;e_1 \rightsquigarrow N_2;H_2;e_2$$

Context compatibility
$$\frac{N_1; H_1; e_1 \rightsquigarrow N_2; H_2; e_2}{N_1; H_1; E[e_1] \rightsquigarrow N_2; H_2; E[e_2]}$$

Primitive reductions

$$\begin{array}{lll}
N;H;\ (\lambda x.e)\ v & \rightsquigarrow\ N;H;\ e[x{:=}v] & \\
N;H;\ \textbf{if true then}\ e_1\ \textbf{else}\ e_2 & \rightsquigarrow\ N;H;\ e_1 & \\
N;H;\ \textbf{if false then}\ e_1\ \textbf{else}\ e_2 & \rightsquigarrow\ N;H;\ e_2 & \\
N;H;\ \underline{\lambda}x.\ e & \rightsquigarrow\ (y,N);H;\ \underline{\lambda}y.\ e[x{:=}\langle y\rangle] & y \notin N \\
N;H;\ \underline{\lambda}y.\langle e\rangle & \rightsquigarrow\ N;H;\ \langle \lambda y.e\rangle & \\
N;H;\ \textbf{ref}\ v & \rightsquigarrow\ N;(l{:}v,H);\ l & l \notin \mathrm{dom}H \\
N;(l{:}v,H);\ !\ l & \rightsquigarrow\ N;(l{:}v,H);\ v & \\
N;(l{:}v,H);\ l\ :=\ v_2 & \rightsquigarrow\ N;(l{:}v_2,H);\ v_2 &
\end{array}$$

Fig. 3. Dynamic semantics of <NJ>: reductions $e_1 \rightsquigarrow e_2$. The reductions involving (code-generating) constants are defined in Fig. 4.

$$\begin{array}{lll}
\underline{\mathsf{cint}}\ n & \rightsquigarrow & \langle n\rangle \\
\underline{\mathsf{cbool}}\ b & \rightsquigarrow & \langle b\rangle \\
\langle e_1\rangle\ \underline{+}\ \langle e_2\rangle & \rightsquigarrow & \langle e_1 + e_2\rangle \\
\langle e_1\rangle\ \underline{=}\ \langle e_2\rangle & \rightsquigarrow & \langle e_1 = e_2\rangle \\
\langle e_1\rangle\ \underline{@}\ \langle e_2\rangle & \rightsquigarrow & \langle e_1\ e_2\rangle \\
\underline{\mathsf{fix}}\ \langle e\rangle & \rightsquigarrow & \langle\ \mathsf{fix}\ e\rangle
\end{array}
\qquad
\begin{array}{lll}
\underline{\mathsf{cif}}\ \langle e_1\rangle\ \langle e_2\rangle\ \langle e_3\rangle & \rightsquigarrow & \langle\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\rangle \\
\mathsf{fix}\ (\lambda f.\ e) & \rightsquigarrow & e[f{:=}\mathsf{fix}\ (\lambda f.\ e)] \\
\underline{\textbf{ref}}\ \langle e\rangle & \rightsquigarrow & \langle\textbf{ref}\ e\rangle \\
\underline{!}\ \langle e\rangle & \rightsquigarrow & \langle !e\rangle \\
\langle e_1\rangle\ \underline{:=}\ \langle e_2\rangle & \rightsquigarrow & \langle e_1 := e_2\rangle
\end{array}$$

Fig. 4. Constant (code-generating) reductions

The full dynamic semantics of <NJ> thus deals with reductions between *configurations*, made of the name and location heaps, and an expression, see Fig. 3. We will often elide the heaps when presenting reductions, especially in examples. As an illustration, the following reductions show the evaluation of a sample imperative code, and the generation of the imperative code:

$$\begin{array}{l}
N;H;\textbf{let}\ r = \textbf{ref}\ (2{+}3)\ \textbf{in}\ r := 0;\ !r \rightsquigarrow^* \\
\quad N;(l{:}5,H);\ l := 0;\ !l \rightsquigarrow^* N;(l{:}0,H);!l \rightsquigarrow N;(l{:}0,H);0 \\
\mathsf{clet}\ r = \underline{\textbf{ref}}\ (\%2 \underline{+} \%3)\ \textbf{in}\ \mathsf{clet}\ z = r \underline{:=} \%0\ \textbf{in}\ \underline{!}r \equiv \\
\quad (\underline{\lambda}r.\ (\underline{\lambda}z.\ \underline{!}r)\ \underline{@}\ r \underline{:=} \%0) \underline{@}\ \underline{\textbf{ref}}\ (\%2 \underline{+} \%3) \rightsquigarrow^* \\
\quad (\underline{\lambda}y.\ (\underline{\lambda}u.\ \underline{!}\langle y\rangle)\ \underline{@}\ \langle y\rangle \underline{:=} \%0) \underline{@}\ \underline{\textbf{ref}}\ (\%2 \underline{+} \%3) \rightsquigarrow^* \\
\quad (\underline{\lambda}y.\ \langle \lambda u.\ !y\rangle\ \underline{@}\ \langle y := 0\rangle) \underline{@}\ \underline{\textbf{ref}}\ (\%2 \underline{+} \%3) \rightsquigarrow^* \\
\quad \langle(\lambda y.\ (\lambda u.\ !y)\ (y := 0))\ (\textbf{ref}\ (2 + 3))\rangle
\end{array}$$

where we used $\mathsf{clet}\ x = e_1\ \textbf{in}\ e_2$ to stand for $(\underline{\lambda}x.\ e_2)\ \underline{@}\ e_1$. In the first example, l denotes a fresh location. We elided the heaps in the second example.

So far, the lambda-calculus fragment of <NJ>, the code generating and the reference cell fragments looked like orthogonal extensions. There is one part of the semantics where they interact non-trivially. It has to do with generating functions and using reference cells to store open code. The following is an ex-

ample of how *not* to use reference cells to store open code: it is the infamous scope-extrusion example.

$$\begin{array}{lll}
\mathsf{N};\mathsf{H};\mathbf{let}\ \mathsf{r} = \mathbf{ref}\ \%0\ \mathbf{in}\ (\underline{\lambda}\mathsf{x}.\ \mathsf{r} := \mathsf{x});\ !\mathsf{r} & \rightsquigarrow & \mathsf{N};(\mathsf{l}:\langle 0\rangle,\mathsf{H});\ (\underline{\lambda}\mathsf{x}.\ \mathsf{l} := \mathsf{x});\ !\mathsf{l} \quad \rightsquigarrow \\
(\mathsf{y},\mathsf{N});(\mathsf{l}:\langle 0\rangle,\mathsf{H});\ (\underline{\lambda}\mathsf{y}.\ \mathsf{l} := \langle \mathsf{y}\rangle);\ !\mathsf{l} & \rightsquigarrow & (\mathsf{y},\mathsf{N});(\mathsf{l}:\langle \mathsf{y}\rangle,\mathsf{H}):\ (\underline{\lambda}\mathsf{y}.\ \langle \mathsf{y}\rangle);\ !\mathsf{l} \quad \rightsquigarrow \\
(\mathsf{y},\mathsf{N});(\mathsf{l}:\langle \mathsf{y}\rangle,\mathsf{H});\ \langle \lambda \mathsf{y}.\ \mathsf{y}\rangle;\ !\mathsf{l} & \rightsquigarrow & \\
(\mathsf{y},\mathsf{N});(\mathsf{l}:\langle \mathsf{y}\rangle,\mathsf{H});\ !\mathsf{l} & \rightsquigarrow & (\mathsf{y},\mathsf{N});(\mathsf{l}:\langle \mathsf{y}\rangle,\mathsf{H});\ \langle \mathsf{y}\rangle
\end{array}$$

When building the functions's body we store the code with the yet-to-be-bound variable y in the reference cell. After the function is constructed we retrieve from the reference cell the code with what is by now the unbound variable y. We have just seen the most blatant example of scope extrusion; alas, there are also subtle, and hence far more dangerous cases; we discuss them in §3.1.

Our dynamic semantics is really non-chalant about unbound future-stage variables, treating them essentially as constants. To be pedantic, y in the result of the scope-extrusion example is bound, technically: it occurs in the name heap. Real staged languages such as MetaOCaml and Scala-Virtualized [14] likewise allow unbound variables to appear in code values (in case of MetaOCaml, for a short interval). We will show in the next section that a well-typed `<NJ>` program never generates code with unbound variables. The scope-extrusion program above does not type-check.

Storing open code in reference cells has many legitimate uses. Here we show one simple example. It is again the power function, but now with reference cells. Merely computing x^n looks in `<NJ>` as

$$\begin{array}{l}
\lambda \mathsf{n}.\lambda \mathsf{x}.\ \mathbf{let}\ \mathsf{r} = \mathbf{ref}\ 1\ \mathbf{in} \\
\quad \mathsf{fix}\ (\lambda \mathsf{f}.\lambda \mathsf{n}.\ \mathbf{if}\ \mathsf{n} = 0\ \mathbf{then}\ 0\ \mathbf{else}\ (\mathsf{r} := !\mathsf{r} * \mathsf{x};\ \mathsf{f}\ (\mathsf{n}-1)))\ \mathsf{n};\ !\mathsf{r}
\end{array}$$

To obtain the code for computing x^n for a fixed n, we turn the above program into the generator, in a rather straightforward way:

$$\begin{array}{l}
\mathbf{let}\ \mathsf{body} = \lambda \mathsf{n}.\lambda \mathsf{x}.\ \mathbf{let}\ \mathsf{r} = \mathbf{ref}\ \%1\ \mathbf{in} \\
\quad \mathsf{fix}\ (\lambda \mathsf{f}.\lambda \mathsf{n}.\ \mathbf{if}\ \mathsf{n} = 0\ \mathbf{then}\ 0\ \mathbf{else}\ (\mathsf{r} := !\mathsf{r}\ \underline{*}\ \mathsf{x};\ \mathsf{f}\ (\mathsf{n}-1)))\ \mathsf{n};\ !\mathsf{r} \\
\mathbf{in}\ \lambda \mathsf{n}.\ \underline{\lambda}\mathsf{x}.\ \mathsf{body}\ \mathsf{n}\ \mathsf{x}
\end{array}$$

Applying the result to, say, 3 produces, as before, $\langle \lambda \mathsf{y}.\ \mathsf{y} * \mathsf{y} * \mathsf{y} * 1\rangle$. The reference cell r accumulates progressively longer code for the product, containing multiple occurrences of the free variable y, to be bound at the end. §4 shows more interesting, realistic example of reference cells in code generators, of assertion insertion.

3 Type System

Fig. 1 also defines the syntax of types t, which include the standard, base types of int and bool, the arrow (function) type and the reference type t **ref**. Non-standard is the code-type $\langle \mathsf{t}\rangle^{\gamma}$, containing the so-called classifier γ – similar in intent, but more precise than the environment classifier of [15], as mentioned in the Introduction. One may think of the classifier γ as a type-level representation, or 'name', of a Level-1 variable – although strictly speaking a classifier represents a binding environment. We delay the further discussion of classifiers

till §3.2, after we explained the typing rules that govern classifiers, the partial order on classifiers and classifier subtyping. Since <NJ> is a two-level system – the generated code does not contain any code generating expressions – we distinguish level 1 types t^1 from level 0 types t: the former omits code types. To relieve the notation burden, however, we will often use the same meta-variable t for both sorts of types, using t^1 only where necessary for disambiguation.

Level indication L	either empty or γ
Environment	$\Gamma ::= [] \mid \Gamma, \gamma \mid \Gamma, (\gamma_1 \succ \gamma_2) \mid \Gamma, (\mathsf{x}{:}\mathsf{t})^L$
Heap typing	$\Theta ::= [] \mid \Theta, (\mathsf{l}{:}\mathsf{t})$
Name heap typing	$\Upsilon ::= [] \mid \Upsilon, \gamma \mid \Upsilon, (\gamma_1 \succ \gamma_2) \mid \Upsilon, (\mathsf{y}{:}\mathsf{t}^1)^\gamma$
Judgement	$\Upsilon;\Theta;\Gamma \vdash^L \mathsf{e}{:}\ \mathsf{t}$

Fig. 5. Judgements, environments, classifiers

Figure 5 defines judgements and their components. The main typing judgement – the expression e has the type t at the level L – has the form $\Upsilon;\Theta;\Gamma \vdash^L \mathsf{e}{:}\ \mathsf{t}$. Here, Γ is the standard environment, an ordered sequence associating types with free variables in an expression. Free variables in e^1 expressions (that is, expressions within brackets) are Level-1 free variables; their associations $(\mathsf{y}{:}\mathsf{t}^1)^\gamma$ in Γ are annotated with the classifier γ. Besides the free variable bindings, Γ also contains classifiers γ and classifier subtyping witnesses $\gamma_1 \succ \gamma_2$ to be explained shortly. Υ and Θ are essentially the typings of the name and location heaps. Θ is indeed a finite map from locations to types; Υ on the other hand, has more structure. It is an ordered sequence. It contains the classifier γ for each name in N. Like Γ, it also contains the types associated with each name (Level-1 variable) and the classifier subtyping witnesses. One may think of Γ as a local type environment and Υ as a 'global' one. The initial Υ contains only the pre-defined classifier γ_0. We use the standard $\in$ notation to assert that Γ or Υ sequences contain a particular element. In addition, we write $\mathsf{l} \in \Theta$ to say the location is in the domain of the finite map Θ. The notation $\mathsf{b} \in (\Gamma \uplus \Upsilon)$ means that some binding b is an element of Γ, or else it is an element of Υ (note the asymmetry).

Some judgements are generic, so we use superscript L that stands for either empty or a classifier. If L is not empty, then, strictly speaking, the judgement should be written as $\Upsilon;[];\Gamma \vdash^\gamma \mathsf{e}^1{:}\ \mathsf{t}^1$, meaning that only a subset of expressions (and types) are allowed at level 1. In particular, locations cannot appear at Level 1 [5]: normally locations result from evaluating expressions **ref** e; although such expressions may appear in the generated code, they remain unevaluated. There are no code combinators in <NJ> that could produce the value $\langle \mathsf{l} \rangle$. A substitution cannot insert a location either, since the generated code cannot be substituted into. Therefore, the heap typing Θ is irrelevant in such judgements. We will almost always drop the superscript in e^1 and t^1 (we keep it in the rule (Code) as reminder).

[5] If we generate code for later use, e.g., as a library of specialized algorithms, it makes no sense for the generated code to contain pointers into the generator's heap. By the time the produced code is run, the generator will be long gone. Although shared heap may be useful in run-time-code specialization, none of the staged calculi to our knowledge consider this case.

$$\frac{}{\vdash [] \ \mathsf{ok}} \qquad \frac{\vdash \Upsilon \ \mathsf{ok} \quad \gamma \notin \Upsilon}{\vdash \Upsilon, \gamma \ \mathsf{ok}} \qquad \frac{\vdash \Upsilon \ \mathsf{ok} \quad \gamma_1 \in \Upsilon \quad \gamma_2 \in \Upsilon}{\vdash \Upsilon, \gamma_1 \succ \gamma_2 \ \mathsf{ok}}$$

$$\frac{\vdash \Upsilon \ \mathsf{ok} \quad \gamma \in \Upsilon \quad \mathsf{y} \notin \Upsilon \quad (\mathsf{y}':\mathsf{t}')^{\gamma} \notin \Upsilon \text{ (i.e., } \Upsilon \text{ has no other binding marked by } \gamma)}{\vdash \Upsilon, (\mathsf{y}:\mathsf{t}^1)^{\gamma} \ \mathsf{ok}}$$

$$\frac{\forall\ \mathsf{l} \in \Theta \text{ such that } \Theta(\mathsf{l}) \text{ is } \langle \mathsf{t} \rangle^{\gamma}.\ \gamma \in \Upsilon}{\Upsilon \vdash \Theta \ \mathsf{ok}}$$

$$\frac{}{\Upsilon \vdash [] \ \mathsf{ok}} \qquad \frac{\Upsilon \vdash \Gamma \ \mathsf{ok} \quad \gamma \notin \Upsilon \quad \gamma \notin \Gamma}{\Upsilon \vdash \Gamma, \gamma \ \mathsf{ok}} \qquad \frac{\Upsilon \vdash \Gamma \ \mathsf{ok} \quad \gamma_1 \in (\Gamma \uplus \Upsilon) \quad \gamma_2 \in (\Gamma \uplus \Upsilon)}{\Upsilon \vdash \Gamma, (\gamma_1 \succ \gamma_2) \ \mathsf{ok}}$$

$$\frac{\Upsilon \vdash \Gamma \ \mathsf{ok} \quad \gamma \in (\Gamma \uplus \Upsilon)}{\Upsilon \vdash \Gamma, (\mathsf{x}:\langle \mathsf{t}^1 \rangle^{\gamma}) \ \mathsf{ok}} \qquad \frac{\Upsilon \vdash \Gamma \ \mathsf{ok} \quad \gamma \in (\Gamma \uplus \Upsilon)}{\Upsilon \vdash \Gamma, (\mathsf{y}:\mathsf{t}^1)^{\gamma} \ \mathsf{ok}}$$

Fig. 6. Well-formedness of environments and heap typings $\vdash \Upsilon \ \mathsf{ok}$, $\Upsilon \vdash \Theta \ \mathsf{ok}$, $\Upsilon \vdash \Gamma \ \mathsf{ok}$

Figure 6 states the well-formedness constraints on the environments and heap typings, which can be summarized as the absence of duplicates and the classifiers being defined before use. It becomes clear that each Level-1 variable binding recorded in the global Υ or local Γ environment has its own classifier. Indeed, a classifier acts as a type-level 'name' of a Level-1 variable. To ease the notation, hereafter we shall assume well-formedness of all environments and heap typings. We write Γ,Γ' and Υ,Υ' for the concatenation of two sequences such that the result must be well-formed.

$$\frac{}{\Upsilon;\Theta;\Gamma \vdash \mathsf{c}:\mathsf{tc}}\ \text{Const} \qquad \frac{(\mathsf{x}:\mathsf{t})^{L} \in (\Gamma \uplus \Upsilon)}{\Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{x}:\mathsf{t}}\ \text{Var} \qquad \frac{\Upsilon\,;[];\Gamma \vdash^{\gamma} \mathsf{e}^1:\mathsf{t}^1}{\Upsilon;\Theta;\Gamma \vdash \langle \mathsf{e}^1 \rangle:\langle \mathsf{t}^1 \rangle^{\gamma}}\ \text{Code}$$

$$\frac{(\mathsf{l}:\mathsf{t}) \in \Theta}{\Upsilon;\Theta;\Gamma \vdash \mathsf{l}:\mathsf{t}}\ \text{Loc} \qquad \frac{\Upsilon;\Theta;\Gamma \vdash \mathsf{e}:\langle \mathsf{t} \rangle^{\gamma_1} \quad \Upsilon;\Gamma \models \gamma_2 \succ \gamma_1}{\Upsilon;\Theta;\Gamma \vdash \mathsf{e}:\langle \mathsf{t} \rangle^{\gamma_2}}\ \text{Sub0}$$

$$\frac{\Upsilon;\Theta;\Gamma \vdash^{\gamma_1} \mathsf{e}:\mathsf{t} \quad \Upsilon;\Gamma \models \gamma_2 \succ \gamma_1}{\Upsilon;\Theta;\Gamma \vdash^{\gamma_2} \mathsf{e}:\mathsf{t}}\ \text{Sub1} \qquad \frac{\Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}_1:\mathsf{t}_1 \to \mathsf{t}_2 \quad \Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}_2:\mathsf{t}_1}{\Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}_1\ \mathsf{e}_2:\mathsf{t}_2}\ \text{App}$$

$$\frac{\Upsilon;\Theta;(\Gamma, (\mathsf{x}:\mathsf{t}_1)^{L}) \vdash^{L} \mathsf{e}:\mathsf{t}_2}{\Upsilon;\Theta;\Gamma \vdash^{L} \lambda \mathsf{x}.\mathsf{e}:\mathsf{t}_1 \to \mathsf{t}_2}\ \text{Abs}$$

$$\frac{\Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}:\mathsf{bool} \quad \Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}_1:\mathsf{t} \quad \Upsilon;\Theta;\Gamma \vdash^{L} \mathsf{e}_2:\mathsf{t}}{\Upsilon;\Theta;\Gamma \vdash^{L} \textbf{if } \mathsf{e} \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2:\mathsf{t}}\ \text{If}$$

$$\frac{\gamma \in (\Gamma \uplus \Upsilon) \quad \gamma_1 \notin (\Gamma \uplus \Upsilon) \quad \Upsilon;\Theta;(\Gamma, \gamma_1, (\gamma_1 \succ \gamma), (\mathsf{x}:\langle \mathsf{t}_1 \rangle^{\gamma_1})) \vdash \mathsf{e}:\langle \mathsf{t}_2 \rangle^{\gamma_1}}{\Upsilon;\Theta;\Gamma \vdash \underline{\lambda}\mathsf{x}.\mathsf{e}:\langle \mathsf{t}_1 \to \mathsf{t}_2 \rangle^{\gamma}}\ \text{CAbs}$$

$$\frac{\Upsilon = \Upsilon', \gamma_1, (\gamma_1 \succ \gamma), (\mathsf{y}:\mathsf{t}_1)^{\gamma_1}, \Upsilon'' \quad \forall\ \gamma_2.\ \Upsilon \models \gamma_1 \succ \gamma_2 \textbf{ and } \gamma_2 \neq \gamma_1 \text{ imply } \Upsilon \models \gamma \succ \gamma_2\ \ \Upsilon;\Theta;[] \vdash \mathsf{e}:\ \langle \mathsf{t}_2 \rangle^{\gamma_1}}{\Upsilon;\Theta;[] \vdash \underline{\lambda}\mathsf{y}.\mathsf{e}:\ \langle \mathsf{t}_1 \to \mathsf{t}_2 \rangle^{\gamma}}\ \text{IAbs}$$

Fig. 7. Type system: typing of expressions

The typing of expressions is presented in Fig. 7 whereas Fig. 9 defines the typing of heaps. Most of the type system is standard. The rule (Const) uses the types of constants tc, given in Fig. 2. We abuse the notation and treat, for

type-checking purposes, constant expressions such as $\mathsf{c}_2\ \mathsf{e}_1\ \mathsf{e}_2$ as applications to c_2, although c_2 is not an expression per se. The rules (Sub0) and (Sub1) rely on the partial order on classifiers specified in Fig. 8 in the straightforward way: $\Upsilon,\Gamma \models \gamma_2 \succ \gamma_1$ if either $\gamma_2 \succ \gamma_1$ literally occurs in the environments as a witness, or can be derived by reflexivity and transitivity.

$$\frac{}{\Upsilon;\Gamma \models \gamma \succ \gamma} \qquad \frac{(\gamma_2 \succ \gamma_1) \in (\Gamma \uplus \Upsilon)}{\Upsilon;\Gamma \models \gamma_2 \succ \gamma_1} \qquad \frac{\Upsilon;\Gamma \models \gamma_1 \succ \gamma_2 \qquad \Upsilon;\Gamma \models \gamma_2 \succ \gamma_3}{\Upsilon;\Gamma \models \gamma_1 \succ \gamma_3}$$

Fig. 8. Partial order on classifiers $\Upsilon;\Gamma \models \gamma_1 \succ \gamma_2$

$$\frac{\forall\ \mathsf{y} \in \mathsf{N}.\ (\mathsf{y}{:}\mathsf{t})^{\gamma} \in \Upsilon}{\Upsilon \vdash \mathsf{N}} \qquad \frac{\forall\ (\mathsf{l}{:}\mathsf{v}) \in \mathsf{H}.\ \Upsilon;\Theta;[] \vdash \mathsf{v} : \Theta(\mathsf{l})}{\Upsilon;\Theta \vdash \mathsf{H}}$$

Fig. 9. Type system: typing of heaps $\Upsilon \vdash \mathsf{N}$ and $\Upsilon;\Theta \vdash \mathsf{H}$

The most interesting are the rules (CAbs) and (IAbs). To explain them and to illustrate the type system, we show two sample typing derivations. The first deals with the term $\underline{\lambda}\mathsf{x}_1.\underline{\lambda}\mathsf{x}_2.\mathsf{x}_1 \mathbin{\underline{+}} \mathsf{x}_2$ – generating the curried addition function – in the initial environment, in which Υ contains only the predefined classifier γ_0, and Θ and Γ are empty. In the following derivation, Γ_2 stands for γ_1, $(\gamma_1 \succ \gamma_0)$, $(\mathsf{x}_1{:}\langle\mathsf{int}\rangle^{\gamma_1})$, γ_2, $(\gamma_2 \succ \gamma_1)$, $(\mathsf{x}_2{:}\langle\mathsf{int}\rangle^{\gamma_2})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Upsilon;\Theta;\Gamma_2 \vdash \mathsf{x}_1\colon \langle\mathsf{int}\rangle^{\gamma_1}} \quad \Upsilon;\Gamma_2 \models \gamma_2 \succ \gamma_1}{\Upsilon;\Theta;\Gamma_2 \vdash \mathsf{x}_1\colon \langle\mathsf{int}\rangle^{\gamma_2}} \quad \dfrac{}{\Upsilon;\Theta;\Gamma_2 \vdash \mathsf{x}_2\colon \langle\mathsf{int}\rangle^{\gamma_2}}}{\Upsilon;\Theta;\Gamma_2 \vdash \mathsf{x}_1 \mathbin{\underline{+}} \mathsf{x}_2\colon \langle\mathsf{int}\rangle^{\gamma_2}}}{\Upsilon;\Theta;(\gamma_1,\ (\gamma_1 \succ \gamma_0),\ (\mathsf{x}_1{:}\langle\mathsf{int}\rangle^{\gamma_1})) \vdash \underline{\lambda}\mathsf{x}_2.\ \mathsf{x}_1 \mathbin{\underline{+}} \mathsf{x}_2\colon \langle\mathsf{int}{\to}\mathsf{int}\rangle^{\gamma_1}}}{\Upsilon;\Theta;[] \vdash \underline{\lambda}\mathsf{x}_1.\underline{\lambda}\mathsf{x}_2.\ \mathsf{x}_1 \mathbin{\underline{+}} \mathsf{x}_2 : \langle\mathsf{int}{\to}\mathsf{int}{\to}\mathsf{int}\rangle^{\gamma_0}}$$

The side-conditions of (CAbs) tell that the classifiers γ_1 and γ_2 are 'fresh'. §3.1 shows another attempted (but not completed) derivation, in case of scope extrusion. The second derivation is for the expression $\underline{\lambda}\mathsf{y}_1.\underline{\lambda}\mathsf{x}_2.\ \langle\mathsf{y}_1\rangle \mathbin{\underline{+}} \mathsf{x}_2$, which results from the one-step reduction of the expression in the previous derivation. Now, Υ_1 stands for γ_0, γ_1, $\gamma_1 \succ \gamma_0$, $(\mathsf{y}_1{:}\mathsf{int})^{\gamma_1}$ and Γ_2 for $\gamma_2,\gamma_2 \succ \gamma_1,(\mathsf{x}_2{:}\langle\mathsf{int}\rangle^{\gamma_2})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Upsilon_1;\Theta;\Gamma_2 \vdash^{\gamma_1} \mathsf{y}_1\colon \mathsf{int}}}{\Upsilon_1;\Theta;\Gamma_2 \vdash \langle\mathsf{y}_1\rangle\colon \langle\mathsf{int}\rangle^{\gamma_1}} \quad \Upsilon_1;\Gamma_2 \models \gamma_2 \succ \gamma_1}{\Upsilon_1;\Theta;\Gamma_2 \vdash \langle\mathsf{y}_1\rangle\colon \langle\mathsf{int}\rangle^{\gamma_2}} \quad \dfrac{}{\Upsilon_1;\Theta;\Gamma_2 \vdash \mathsf{x}_2\colon \langle\mathsf{int}\rangle^{\gamma_2}}}{\Upsilon_1;\Theta;\Gamma_2 \vdash \langle\mathsf{y}_1\rangle \mathbin{\underline{+}} \mathsf{x}_2\colon \langle\mathsf{int}\rangle^{\gamma_2}}}{\Upsilon_1;\Theta;[] \vdash \underline{\lambda}\mathsf{x}_2.\ \langle\mathsf{y}_1\rangle \mathbin{\underline{+}} \mathsf{x}_2\colon \langle\mathsf{int}{\to}\mathsf{int}\rangle^{\gamma_1}}}{\Upsilon_1;\Theta;[] \vdash \underline{\lambda}\mathsf{y}_1.\underline{\lambda}\mathsf{x}_2.\ \langle\mathsf{y}_1\rangle \mathbin{\underline{+}} \mathsf{x}_2 : \langle\mathsf{int}{\to}\mathsf{int}{\to}\mathsf{int}\rangle^{\gamma_0}}$$

It should be clear, already from (IAbs) in fact, that $\underline{\lambda}\mathsf{y}.$ is not really a binding form. The environment Γ in (IAbs) is empty since $\underline{\lambda}\mathsf{y}.\mathsf{e}$ shows up only during evaluation and it is not a value.

Proposition 1 (Canonical Forms). *The only values of base types* int *and* bool *are zero-ary constants (numerals and booleans, respectively). Values of reference*

types t **ref** *are locations. Values of code types are all bracketed expressions* $\langle e \rangle$ *and of the function types* $t_1 \to t_2$ *are abstractions* $\lambda x.e$.

Although constants of arity 1 and above also have function types (see Fig. 2), not applied to the right number of arguments they are not regarded as expressions.

Proposition 2 (Weakening). *If* $\Upsilon;\Theta;\Gamma \vdash^L e{:}t$, $\Upsilon \vdash N$, *and* $\Upsilon;\Theta \vdash H$ *hold, so do* $(\Upsilon,\Upsilon');(\Theta,\Theta');(\Gamma,\Gamma') \vdash^L e{:}t$ *and* $(\Upsilon,\Upsilon') \vdash N$ *and* $(\Upsilon,\Upsilon');(\Theta,\Theta') \vdash H$.

Recall that comma denotes concatenation that preserves well-formedness; which implies Θ and Θ' are disjoint. The proof is straightforward.

Theorem 1 (Subject Reduction). *If* $\Upsilon;\Theta;[] \vdash e{:}\ t$, $\Upsilon \vdash N$, $\Upsilon;\Theta \vdash H$, *and* $N;H;e \rightsquigarrow N';H';e'$, *then* $\Upsilon';\Theta';[] \vdash e'{:}\ t$, $\Upsilon' \vdash N'$, $\Upsilon';\Theta' \vdash H'$, *for some* Υ' *and* Θ' *that are the extensions of the corresponding unprimed things.*

We outline the proof in Appendix A.

Theorem 2 (Progress). *If* $\Upsilon;\Theta;[] \vdash e{:}t$, $\Upsilon \vdash N$ *and* $\Upsilon;\Theta \vdash H$, *then either* e *is a value or there are* N',H' *and* e' *such that* $N;H;e \rightsquigarrow N';H';e'$.

The proof is the easy consequence of the canonical forms lemma. For example, if the last rule in the derivation of $\Upsilon;\Theta;[] \vdash \mathsf{e}{:}\mathsf{t}$ is (IAbs), then e must have the form $\underline{\lambda}\mathsf{y}.\mathsf{e}'$ for some e', where e' must itself be typeable in the same Υ and Θ. By induction hypothesis, e' either reduces, or is a value. In the latter case, by the canonical forms lemma, it should be of the form $\langle \mathsf{e}_2 \rangle$ for some e_2 – meaning $\underline{\lambda}\mathsf{y}.\langle \mathsf{e}_2 \rangle$ can reduce.

Corollary 1. *If* $([],\gamma_0);[];[] \vdash e{:}\langle t \rangle^{\gamma_0}$ *and* $[];[];e \rightsquigarrow N;H;v$ *then* v *has the form* $\langle e_1 \rangle$ *and* $([],\gamma_0);[];[] \vdash^{\gamma_0} e_1{:}t$.

That is, if a well-typed program of the type $\langle \mathsf{t} \rangle^{\gamma_0}$ terminates it generates the code well-typed in the empty environment. The generated code hence has no unbound variables.

3.1 Scope Extrusion

When generating the body of a function, its formal argument is available as a code value – as the free variable. Scope extrusion occurs when that open code value is used outside the dynamic scope of the function generator and hence the free variable can never be properly bound. Although the error is obvious once we attempt to compile the generated code, it is not at all obvious what part of the generator is responsible. Debugging generated code is very difficult in general. We now demonstrate how `<NJ>` prevents scope extrusion.

We start with the example of blatant scope extrusion, from §2:

```
let r = ref %0 in (λx. r := x); !r
```

We have seen that its evaluation indeed produces the code with an unbound variable. The example does not type check however. Specifically, the type error occurs not when the open code is retrieved from the reference cell r at the end. Rather, the generator of the function body, specifically, r := x fails to type-check. Here is the attempt at the derivation, where we assumed $\gamma \in (\Gamma \uplus \Upsilon)$ and so is γ_1 (which may be the same as γ). We take Γ_2 to be Γ,(r:$\langle$int$\rangle^{\gamma_1}$ **ref**),γ_2,$\gamma_2 \succ \gamma$,(x:$\langle$int$\rangle^{\gamma_2}$) where γ_2 is fresh.

$$\dfrac{\dfrac{\Upsilon;\Theta;\Gamma_2 \vdash \mathsf{r} := \mathsf{x}\colon \langle \mathsf{int}\rangle^{\gamma_2}}{\Upsilon;\Theta;(\Gamma,(\mathsf{r}\colon\langle \mathsf{int}\rangle^{\gamma_1}\ \mathbf{ref})) \vdash (\underline{\lambda}\mathsf{x}.\ \mathsf{r} := \mathsf{x}) \ :\ \langle \mathsf{int}\to\mathsf{int}\rangle^{\gamma}}}{\Upsilon;\Theta;\Gamma \vdash \mathbf{let}\ \mathsf{r} = \mathbf{ref}\ \underline{\mathsf{cint}}\ 0\ \mathbf{in}\ (\underline{\lambda}\mathsf{x}.\ \mathsf{r} := \mathsf{x}) \ :\ \langle \mathsf{int}\to\mathsf{int}\rangle^{\gamma}}$$

The derivation cannot be completed since r has the type $\langle$int$\rangle^{\gamma_1}$ **ref** but x is of the type $\langle$int$\rangle^{\gamma_2}$ where γ_2 is specifically chosen by (CAbs) to be different from any other classifiers in Γ and Υ, including γ_1.

If such examples were our only worry, a simpler type system would have sufficed. Instead of named classifiers, we would annotate code types with just a natural number: the nesting level of $\underline{\lambda}$. Our blatant example will likewise fail to type-check. The error will be reported later, however, when type checking the last expression !r retrieving the code with the already leaked variable as the program result. The program result must be closed: be at the 0th nesting level. The type system of [3] (extended with reference cells) likewise rejects the blatant example, as was described in that paper. (After all, their type system annotates code types with the typing environment sequence, which is the refinement of the nesting depth.) MetaOCaml also reports the scope extrusion error – when running the program and executing the !r expression. In contrast, `<NJ>` rejects r := x, when merely attempting to leak out the free variable.

Alas, scope extrusion can be subtle. Consider a small modification of the earlier example:

let r = **ref** %0 **in** ($\underline{\lambda}$x. r := x); ($\underline{\lambda}$z. !r)

The simpler type system with mere level counting accepts the code: the free variable leaks out of one binder into another, at the same nesting level of $\underline{\lambda}$. Likewise, the calculus of [3] (extended with reference cells as described therein) will type-check and even run the example, producing the code for the identity function. This is not what one may expect from the generator $\underline{\lambda}$z. !r. Our `<NJ>` rejects r := x in the first part of the example as described earlier: it rejects even an attempt to leak the variable.

Finally, scope extrusion may be harmless, as in the following, yet another variation of the example:

let r = **ref** (λz.z) **in** ($\underline{\lambda}$x. r := (λz. (x; z)); %0); !r

When generating the body of the function, we incorporate the free variable x in the closure λz. (x; z), but in a way that it does not contribute to the result and hence is not reflected in the closure's type, which remains int→int. Technically, the free variable has leaked – but in a useless way, embedded in dead code.

`<NJ>` accepts the latter example. When run, it indeed produces the closure with an unbound variable – which remains typeable since the unbound variable

is still in the global heap N and its classifier in Υ. Such open code must have been dead, however: it cannot be the result of a well-typed generator, since the type of such result would have contained the classifier γ that is different from γ_0. The well-typed generator program must have the type $\langle \mathsf{t} \rangle^{\gamma_0}$. We have seen before that even a fragment, let alone the whole program, that attempts to 'usefully leak' a bound variable will fail to type-check.

Accepting unbound variables in dead code has many precedents. Most region calculi (see [5] and references therein) and their implementations (such as runST monad in Haskell) allow dangling references, provided they are not accessed – that is, remain embedded in essentially dead code.

3.2 Environment Classifiers, Binding Abstractions, and Lexical Scope

As we have seen from §3.1, the key to preventing scope extrusion is annotating the type of a code value with some representation of free variables that may be contained therein. This section discusses a few choices for the representation and the position of `<NJ>` among them as the most abstract while still sufficient to prevent scope extrusion. By free variables we always mean Level-1 free variables: all values and terms produced and evaluated in stage calculi are closed with respect to Level-0 variables.

On one end of the spectrum is annotating the type of a code value with the names of the containing free variables, or the typing environment: the set or the sequence listing the free variables and their types. Taha and Nielsen [15, §1.4] describe many difficulties of this approach (the sheer size of the type being one of them), which makes it hard to implement, and use in practice.

On the other extreme is the most abstract representation of a set of free variables: as a single name (the environment classifier, [15]) or a number, the cardinality of the set. §3.1 showed that this is not sufficient to prevent the scope extrusion, of the devious, most harmful sort.

The approach of [3] also annotates the code types with the type environment; however, by using De Bruijn indices, it avoids many difficulties of the nominal approach, such as freshness constraints, α-renaming, etc. The approach is indeed relatively easy to implement, as the authors have demonstrated. Alas, although preventing blatant scope extrusion, it allows the devious one, as we saw in §3.1.

The representation of [3] is also just too concrete: the code type $\langle \mathsf{int} \rangle^{(\mathsf{int},\mathsf{bool},\mathsf{int})}$ tells not only that the value may contain three free variables with the indices 0, 1 and 2. The type also tells that the int and the bool variables will be bound in that order and there is no free variable to be bound in between. There is no need to know with such exactitude when free variables will be bound. In fact, there is no need to even know their number, to prevent scope extrusion. The concreteness of the representation has the price: the system of [3, §3.3] admits the term, in our notation, $\lambda \mathsf{f}.\underline{\lambda}\mathsf{x}.\underline{\lambda}\mathsf{y}.\mathsf{f}\ \mathsf{y}$, which may, depending on the argument f, generate either $\langle \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{y} \rangle$ or, contrary to any expectation, $\langle \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} \rangle$.

Such a behavior is just not possible in `<NJ>`: consider $\underline{\lambda}\mathsf{x}.\ \mathsf{f}\ \mathsf{x}$ where f is some function on code values. The function receives the code of a Level-1 variable

and is free to do anything with it: discard it, use it once or several times in the code it is building, store in global reference cells, as well as do any other effects, throw exceptions or diverge. Still, we are positive that whatever f may do, if it eventually returns the code that includes the received Level-1 variable, that variable shall be bound by $\underline{\lambda}$x. of our expression – regardless of whatever binders f may introduce. This is what we call 'lexical' scope for Level-1 variables: the property, not present in [7] (by choice) or [3].

<NJ> avoids the problematic 'variable conversions' because it does not exposes in types or at run-time any structure of the Level-1 typing environment. The environment classifier in <NJ> is the type-level representation of the variable name. There is a partial order on classifiers, reflecting the nesting order of the corresponding $\underline{\lambda}$x generators. The relation $\gamma_2 \succ \gamma_1$ tells that the variable corresponding to γ_1 is (to be) introduced earlier than the free variable corresponding to γ_2, with no word on which or how many variables are to be introduced in-between. The code type is annotated not with the set of free variables, not with the set of the corresponding classifiers – but only with the single classifier, the maximal in the set. The type system ensures that there is always the maximal element. To be precise, any free Level-1 variable that may appear within $\underline{\lambda}$y. $\langle \mathsf{e} \rangle : \langle \mathsf{t}_1 \to \mathsf{t}_2 \rangle^{\gamma_2}$ is marked by such a classifier γ_1 that $\gamma_2 \succ \gamma_1$. Therefore, any such variable will be bound by an ancestor of $\underline{\lambda}$y. This is another way to state the property of 'lexical scope' for free variables.

3.3 Classifier Polymorphism

The classifier polymorphism and its importance are best explained on examples. The following generator

$$\lambda x.\, \textbf{let}\ \mathsf{f} = \underline{\lambda} z.\ \underline{\mathsf{cint}}\ \mathsf{x} \mathbin{\underline{+}} \%1 \mathbin{\underline{+}} \mathsf{z}\ \textbf{in let}\ \mathsf{f'} = \underline{\lambda} z'.\ (\underline{\mathsf{cint}}\ \mathsf{x} \mathbin{\underline{+}} \%1) \mathbin{\underline{*}} \mathsf{z'}\ \textbf{in}\ \mathsf{e}$$

contains the repeated code that we would like to factor out, to make the generators clearer and more modular:

$$\lambda x.\, \textbf{let}\ \mathsf{u} = \underline{\mathsf{cint}}\ \mathsf{x} \mathbin{\underline{+}} \%1\ \textbf{in let}\ \mathsf{f} = \underline{\lambda} z.\ \mathsf{u} \mathbin{\underline{+}} \mathsf{z}\ \textbf{in}\ \ \textbf{let}\ \ \mathsf{f'} = \underline{\lambda} z'.\ \mathsf{u} \mathbin{\underline{*}} \mathsf{z'}\ \ \textbf{in}\ \ \mathsf{e}$$

One may worry if the code type-checks: after all, u is used in contexts associated with two distinct classifiers. The example does type-check, thanks to (Sub0) rule: u can be given the type $\langle \mathsf{int} \rangle^{\gamma_0}$, and although $\mathsf{z}{:}\langle \mathsf{int} \rangle^{\gamma_1}$ and $\mathsf{z'}{:}\langle \mathsf{int} \rangle^{\gamma_2}$ are associated with unrelated classifiers, $\gamma_1 \succ \gamma_0$ and $\gamma_2 \succ \gamma_0$ hold.

Alas, the classifier subtyping gets us only that far. It will not help in the more interesting and common example of functions on code values:

$$\lambda x.\, \textbf{let}\ \mathsf{u} = \lambda z.\underline{\mathsf{cint}}\ \mathsf{x} \mathbin{\underline{+}} \mathsf{z}\ \textbf{in}\ \ \textbf{let}\ \ \mathsf{f} = \underline{\lambda} z.\ \mathsf{u}\ \mathsf{z} \mathbin{\underline{+}} \mathsf{z}\ \textbf{in}\ \ \textbf{let}\ \ \mathsf{f'} = \underline{\lambda} z'.\ \mathsf{u}\ \mathsf{z'} \mathbin{\underline{*}} \mathsf{z'}\ \ \textbf{in}\ \ \mathsf{e}$$

where the function u is applied to code values associated with unrelated classifiers. To type-check this example we need to give u the type $\forall\, \gamma.\ \langle \mathsf{int} \rangle^{\gamma} \to \langle \mathsf{int} \rangle^{\gamma}$. Before, γ was used as a (sometimes schematic) constant; now we use it as a classifier variable.

Extending <NJ> with let-bound classifier polymorphism with attendant value restriction is unproblematic and straightforward. In fact, our implementation

already does it, inheriting let-polymorphism from the host language, OCaml. Sometimes we may need more extensions, however.

For example, we may write a generator that introduces an arbitrary, statically unknown number of Level-1 variables, e.g., as let-bound variables to share the results of computed expressions. Such pattern occurs, for example, when specializing dynamic programming algorithms. Appendix B demonstrates the let-sharing on the toy example of specializing the Fibonacci-like function, described in [6, §2.4]. As that paper explains, the generator requires polymorphic recursion – which is well-understood. Both Haskell and OCaml supports it, and hence our implementation of `<NJ>`. Polymorphic recursion also shows in [3].

There are, however, times (not frequent, in our experience) where even more polymorphism is needed. The poster example is the staged eta-function, the motivating example in [15]: λf. $\underline{\lambda}$x. f x, whose type is, $(\langle \mathsf{t}_1\rangle^\gamma \to \langle \mathsf{t}_2\rangle^\gamma) \to \langle \mathsf{t}_1{\to}\mathsf{t}_2\rangle^\gamma$, approximately. The type is not quite right: f accepts the code value that contains a fresh free variable, which comes with a previously unseen classifier. Hence we should assign eta at least the type $(\forall\ \gamma_1.\ \langle \mathsf{t}_1\rangle^{\gamma_1} \to \langle \mathsf{t}_2\rangle^{\gamma_1}) \to \langle \mathsf{t}_1{\to}\mathsf{t}_2\rangle^\gamma$ – the rank-2 type. This is still not quite right: we would like to use eta in the expression such as $\underline{\lambda}$u. eta (λz. u $\underline{+}$ z), where f combines the open code received as argument with some other open code. To type-check this combination we need Υ,$\Gamma \models \gamma_1 \succ \gamma$. Hence the correct type for eta should be

$$\forall\ \gamma.\ (\forall\ \gamma_1 \succ \gamma.\ \langle \mathsf{t}_1\rangle^{\gamma_1} \to \langle \mathsf{t}_2\rangle^{\gamma_1}) \to \langle \mathsf{t}_1{\to}\mathsf{t}_2\rangle^\gamma$$

with the bounded quantification. One is immediately reminded of ML^F. Such bounded quantification is easy to implement, however, by explicit passing of subtyping witnesses (as done in the implementation of the region calculus [5]) Our implementation of `<NJ>` supports it too – and how it cannot: eta is just the first-class form of $\underline{\lambda}$. Thus the practical drawback is the need for explicit type signatures for the sake of the rank-2 type (just as signatures are required in ML^F when the polymorphic argument function is used polymorphically). Incidentally, the original environment classifiers calculus of [15] gives eta the ordinary rank-1 type: here the coarseness of the original classifiers is the advantage. The formal treatment of rank-2 classifier polymorphism is the subject of the future research.

4 Complex Example

To demonstrate the expressiveness of `<NJ>`, we show a realistic example of assert-insertion – exactly the same example that was previously written in Staged-Haskell. The latter is the practical Haskell code-generation library, too complex to reason about formally and prove correctness. The example was explained in detail in [6]; therefore, we elide many explanations here.

For the sake of the example, we add the following constants to `<NJ>`:

/	: int → int → int	**assert**	: bool → bool
$\underline{/}$	: $\langle$int$\rangle^\gamma \to \langle$int$\rangle^\gamma \to \langle$int$\rangle^\gamma$	assertPos	: $\langle$int$\rangle^\gamma \to \langle$t$\rangle^\gamma \to \langle$t$\rangle^\gamma$

The first two are the integer division and the corresponding code combinator; **assert** e returns the result of the boolean expression, if it is true. Otherwise, it

crashes the program. The constant assertPos is the corresponding combinator, with the reduction rule assertPos $\langle e_1 \rangle$ $\langle e_2 \rangle \rightsquigarrow \langle$**assert** $(e_1{>}0)$; $e_2\rangle$.

The goal is to implement the guarded division, which makes sure that the divisor is positive before attempting the operation. The naive version

```
let guarded_div = λx.λy. assertPos y (x / y)
```

to be used as

```
λy. complexExp + guarded_div %10 y
```

produces $\langle \lambda$x. complexExp + (**assert** (x>0); (10 / x))$\rangle$. The result is hardly satisfactory: we check the divisor right before the division. If it is not positive, the time spent computing complexExp is wasted. If the program is going to end up in error, we had rather it end sooner than much later.

The solution is explained in [6], implemented in StagedHaskell and is reproduced below in `<NJ>`. Intuitively, we first reserve the place where it is appropriate to place assertions, which is typically right at the beginning of a function. As we go on generating the body of the function, we determine the assertions to insert and accumulate them in a mutable 'locus'. Finally, when the body of the function is generated, we retrieve the accumulated assertion code and prepend it to the body. The function add_assert below accumulates the assertions; assert_locus allocates the locus at the beginning and applies the accumulated assertions at the end.

```
let assert_locus = λf.
  let r = ref (λx.x)    in let c = f r in
  let transformer = !r in transformer c

let add_assert locus transformer =
  locus := (let oldtr = !locus in λx. oldtr (transformer x))

let guarded_div = λlocus.λx.λy. add_assert locus (λz. assertPos y z); (x / y)
```

They are to be used as in the example below:

```
λy. assert_locus (λlocus. λz. complexExp + guarded_div locus z y)
```

As we generate the code, the reference cell r within the locus accumulates the transformer (code-to-code function), to be applied to the result. In our example, the code transformer includes open code (embedded within the assertPos expression), which is moved from within the generator of the inner function. The example thus illustrates all the complexities of imperative code generation. The improved generated code

```
⟨λx. assert (x>0); (λy. complexExp + y / x)⟩
```

checks the divisor much earlier: before we started on complexExp, before we even apply the function (λy. complexExp + y / x). If we by mistake switch y and z in guarded_div locus z y, we get a type-error message.

5 Related Work

We thoroughly review the large body of related work in [6]. Here we highlight only the closest connections. First is Template Haskell, which either permits effectful generators but then provides no guarantees by construction; or provides guarantees but permits no effects – the common trade-off. We discuss this issue in detail in [6]. BER MetaOCaml [8] permits any effects and ensures well-scopedness, even in open fragments, using dynamic checks. StagedHaskell and `<NJ>` are designed to prevent scope extrusion even before running the generator.

Safe imperative multi-staged programming has been investigated in [1] and [17]. Safety comes at the expense of expressiveness: e.g., only closed code is allowed to be stored in mutable cells (in the former approach).

We share with [15] the idea of using an opaque label, the environment classifier, to refer to a typing environment. The main advantage of environment classifiers, their imprecision (they refer to infinite sets of environments), is also their drawback. On one hand, they let us specify staged-eta §3.3 without any first-class polymorphism. On the other hand, the imprecision is not enough to safely use effects.

Chen and Xi [3] and Nanevski et al. [10] annotate the code type with the type environment of its free variables. The former relies on the first-order syntax with De Bruijn indices whereas the latter uses higher-order abstract syntax. Although internally Chen and Xi use De Bruijn indices, they develop a pleasant surface syntax a la MetaOCaml (or Lisp's antiquotations). The De Bruijn indices are still there, which may lead to unpleasant surprises, which they discuss in [3, §3.3]. Their type system indeed rejects the blatant example of scope extrusion. Perhaps that is why [3] said that reference cells do not bring in significant complications. However, scope extrusion is much subtler than its well-known example: §3.1 presented a just slightly modified example, which is accepted in Chen and Xi's system, but produces an unexpected result. We refer to [6] for extensive discussion.

One may think that any suitable staged calculus can support reference cells through a state-passing translation. The elaborate side-conditions of our (CAbs) and (IAbs) rules indicate that a straightforward state-passing translation is not going to be successful to ensure type and scope safety.

Staged-calculi of [3, 15] have a special constant `run` to run the generated code. Adding it to `<NJ>` is straightforward.

Our bracketed expressions are the generalization of data constructors of the code data type in the 'single-stage target language' [2, Fig.2]. Our name heap also comes from the same calculus. The latter, unlike `<NJ>`, is untyped, and without effects.

6 Conclusions and Future Work

We have described the first staged calculus `<NJ>` for imperative code generators without ad hoc restrictions – letting us even store open code in reference cells and

retrieve it in a different binding environment. Its sound type system statically assures that the generated code is by construction well-typed and well-scoped, free from unbound or surprisingly bound variables. The calculus has been distilled from StagedHaskell, letting us formally prove the soundness of the latter's approach. The distilled calculus is still capable of implementing StagedHaskell's examples that use mutation.

`<NJ>` has drawn inspiration from such diverse areas as region-based memory management and Natural Deduction. It turns out a vantage point to overview these areas.

`<NJ>` trivially generalizes to effects such as exceptions or IO. It is also easy to extend with new non-binding language forms. (Binding-forms like for-loops can always be expressed via lambda-forms: see Haskell or Scala, for example.) `<NJ>` thus serves as the foundation of real staged programming languages. In fact, it is already implemented as an OCaml library. Although the explicit weakening is certainly cumbersome, it turns out, in our experience, not as cumbersome as we had feared. It is not a stretch to recommend the OCaml implementation of `<NJ>` as a new, safe, staged programming language.

Extension to effects such as delimited control or non-determinism is however non-trivial and is the subject of on-going research. We are also investigating adding first-class bounded polymorphism for classifiers, relating `<NJ>` more precisely to ML^F.

Acknowledgments We thank anonymous reviewers for many helpful comments. This work was partially supported by JSPS KAKENHI Grant Numbers 15K12007, 16K12409, 15H02681.

References

[1] Calcagno, C., Moggi, E., Taha, W.: Closed types as a simple approach to safe imperative multi-stage programming. In: ICALP. pp. 25–36. No. 1853 in LNCS (2000)

[2] Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: GPCE. pp. 57–76. No. 2830 in LNCS (22–25 Sep 2003)

[3] Chen, C., Xi, H.: Meta-programming through typeful code representation. Journal of Functional Programming 15(6), 797–835 (2005)

[4] Davies, R.: A temporal logic approach to binding-time analysis. In: LICS. pp. 184–195 (1996)

[5] Fluet, M., Morrisett, J.G.: Monadic regions. Journal of Functional Programming 16(4–5), 485–545 (2006)

[6] Kameyama, Y., Kiselyov, O., Shan, C.c.: Combinators for impure yet hygienic code generation. Science of Computer Programming 112 (part 2), 120–144 (Nov 2015)

[7] Kim, I.S., Yi, K., Calcagno, C.: A polymorphic modal type system for Lisp-like multi-staged languages. In: POPL. pp. 257–268 (2006)

[8] Kiselyov, O.: The design and implementation of BER MetaOCaml - system description. In: FLOPS. pp. 86–102. No. 8475 in LNCS, Springer (2014)

[9] Le Botlan, D., Rémy, D.: ML^F: Raising ML to the power of System F. In: ICFP. pp. 27–38 (2003)
[10] Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. Transactions on Computational Logic 9(3), 23:1–49 (Jun 2008)
[11] POPL '03: Conference Record of the Annual ACM Symposium on Principles of Programming Languages (2003)
[12] Pottier, F.: Static name control for freshML. In: LICS. pp. 356–365. IEEE Computer Society (2007)
[13] Pouillard, N., Pottier, F.: A fresh look at programming with names and binders. In: ICFP. pp. 217–228. ACM Press, New York (2010)
[14] Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-Virtualized: linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation (Sep.) (2013)
[15] Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL [11], pp. 26–37
[16] Thiemann, P.: Combinators for program generation. Journal of Functional Programming 9(5), 483–525 (1999)
[17] Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: PLDI '10. ACM Press, New York (2010)
[18] Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL [11], pp. 224–235

A Proof outlines: Subject reduction theorem

Lemma 1 (Substitution). *(1) If* $\Upsilon;\Theta;(\Gamma,(x{:}t_1)) \vdash e{:}\ t$ *and* $\Upsilon,\Theta,\Gamma \vdash e_1{:}\ t_1$ *then* $\Upsilon;\Theta;\Gamma \vdash e[x{:=}e_1]{:}\ t$
(2) If $\Upsilon;\Theta;(\Gamma,\gamma_2,\gamma_2{\succ}\gamma_1,\Gamma') \vdash^L e{:}\ t$ *and* $\gamma_1 \in \Upsilon$ *and* $\gamma_2' \notin (\Gamma \uplus \Upsilon)$, *then* $(\Upsilon,\gamma_2',\gamma_2'{\succ}\gamma_1);\Theta,(\Gamma,\Gamma'[\gamma_2{:=}\gamma_2']) \vdash^L e{:}\ t[\gamma_2{:=}\gamma_2']$ *(if* L *was* γ_2 *it is also replaced with* γ_2'*).*

This lemma is proved straightforwardly.

Theorem 3 (Subject Reduction). *If* $\Upsilon;\Theta;[] \vdash e{:}\ t$, $\Upsilon \vdash N$, $\Upsilon;\Theta \vdash H$, *and* $N;H;e \rightsquigarrow N';H';e'$, *then* $\Upsilon';\Theta';[] \vdash e'{:}\ t$, $\Upsilon' \vdash N'$, $\Upsilon';\Theta' \vdash H'$, *for some* Υ' *and* Θ' *that are the extensions of the corresponding unprimed things.*

Proof. We consider a few interesting reductions. The first one is

$$N;H;\underline{\lambda}x.\ e \ \rightsquigarrow\ (N,y);H;\underline{\lambda}\ y.\ e[x{:=}\langle y\rangle],\ y \notin N$$

We are given N' is N,y, H' is H, and $\Upsilon;\Theta;[] \vdash \underline{\lambda}x.e : \langle t_1{\to}t_2\rangle^\gamma$, which means $\gamma \in \Upsilon$ and $\Upsilon;\Theta;(\gamma_2,\gamma_2{\succ}\gamma,(x{:}\langle t_1\rangle^{\gamma_2})) \vdash e{:}\langle t_2\rangle^{\gamma_2}$ for a fresh γ_2. We choose Υ' as $\Upsilon,\gamma_1,\gamma_1{\succ}\gamma,(y{:}t_1)^{\gamma_1}$ where γ_1 is fresh, and Θ' as Θ. Υ' is well-formed and is an extension of Υ. Furthermore, $\Upsilon' \vdash N,y$. By weakening, $\Upsilon' \vdash \Theta$ ok and $\Upsilon';\Theta \vdash H$ if it was for Υ. We only need to show that $\Upsilon';\Theta;[] \vdash \underline{\lambda}y.\ e[x{:=}\langle y\rangle] : \langle t_1{\to}t_2\rangle^\gamma$, which follows by (IAbs) from $\Upsilon';\Theta;[] \vdash e[x{:=}\langle y\rangle] : \langle t_2\rangle^{\gamma_1}$, which in turn follows from from the fact that $\Upsilon';\Theta;[] \vdash \langle y\rangle : \langle t_1\rangle^{\gamma_1}$ and the substitution lemma.

The next reduction is

$$N;H;\underline{\lambda}y.\langle e\rangle \ \rightsquigarrow\ N;H;\langle\lambda y.e\rangle$$

We are given $\Upsilon;\Theta;[] \vdash \underline{\lambda}\mathsf{y}.\langle \mathsf{e}\rangle : \langle \mathsf{t}_1 \to \mathsf{t}_2\rangle^{\gamma}$, $\Upsilon \vdash \mathsf{N}$ and $\Upsilon,\Theta \vdash \mathsf{H}$. Since N and H are unchanged by the reduction, we do not extend Υ and Θ. By inversion of (IAbs) we know that Υ is $\Upsilon',\gamma_1,\gamma_1 \succ \gamma,(\mathsf{y}{:}\mathsf{t}_1)^{\gamma_1},\Upsilon''$ and $\forall\ \gamma_2.\ \Upsilon \models \gamma_1 \succ \gamma_2$ **and** $\gamma_2 \neq \gamma_1$ imply $\Upsilon \models \gamma \succ \gamma_2$ and $\Upsilon;\Theta;[] \vdash \langle \mathsf{e}\rangle : \langle \mathsf{t}_2\rangle^{\gamma_1}$, or, by inversion of (Code) $\Upsilon;[];[] \vdash^{\gamma_1} \mathsf{e} : \mathsf{t}_2$. By weakening, $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma_1} \mathsf{e} : \mathsf{t}_2$. An easy substitution lemma gives us $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma_1} \mathsf{e}' : \mathsf{t}_2$ where e' is $\mathsf{e}[\mathsf{y}{:=}\mathsf{y}']$, keeping in mind that $\Upsilon \models \gamma_1 \succ \gamma$. The crucial step is strengthening. Since we have just substituted away $(\mathsf{y}{:}\mathsf{t}_1)^{\gamma_1}$, which is the only variable with the classifier γ_1 (the correspondence of variable names and classifiers is the consequence of well-formedness), the derivation $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma_1} \mathsf{e}' : \mathsf{t}_2$ has no occurrence of the rule (Var) with L equal to γ_1. Therefore, any subderivation with L being γ_1 must have the occurrence of the (Sub1) rule, applied to the derivation $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma_2} \mathsf{e}'{:}\mathsf{t}'$ where $\Upsilon \models \gamma_1 \succ \gamma_2$ and γ_2 is different from γ_1. The inversion of (IAbs) gave us $\forall\ \gamma_2.\ \Upsilon \models \gamma_1 \succ \gamma_2$ **and** $\gamma_1 \neq \gamma_2$ imply $\Upsilon \models \gamma \succ \gamma_2$. Therefore, we can always replace each such occurrence of (Sub1) with the one that gives us $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma} \mathsf{e}'{:}\mathsf{t}'$. All in all, we build the derivation of $\Upsilon;[];([],(\mathsf{y}'{:}\mathsf{t})^{\gamma}) \vdash^{\gamma} \mathsf{e}' : \mathsf{t}_2$, which gives us $\Upsilon;[];[] \vdash^{\gamma} \lambda\mathsf{y}.\mathsf{e} : \mathsf{t}_1 \to \mathsf{t}_2$ and then $\Upsilon;[];[] \vdash \langle \lambda\mathsf{y}.\mathsf{e}\rangle : \langle \mathsf{t}_1 \to \mathsf{t}_2\rangle^{\gamma}$.

Another interesting case is

$$\mathsf{N};\mathsf{H};\underline{\lambda}\mathsf{y}.\mathsf{E}[\mathbf{ref}\ \langle \mathsf{y}\rangle] \ \rightsquigarrow\ \mathsf{N};(\mathsf{H},\mathsf{l}{:}\langle \mathsf{y}\rangle);\underline{\lambda}\mathsf{y}.\mathsf{E}[\mathsf{l}]$$

Given, $\Upsilon;\Theta;[] \vdash \underline{\lambda}\mathsf{y}.\mathsf{E}[\mathbf{ref}\ \langle \mathsf{y}\rangle] : \langle \mathsf{t}_1 \to \mathsf{t}_2\rangle^{\gamma}$ which means $\Upsilon = \Upsilon'$, γ_1, $\gamma_1 \succ \gamma$, $(\mathsf{y}{:}\mathsf{t}_1)^{\gamma_1}$, Υ''. Take $\Theta' = \Theta,(\mathsf{l}{:}\overline{\langle \mathsf{t}_1\rangle}^{\gamma_1}\ \mathbf{ref})$. It is easy to see that $\Upsilon \vdash \Theta'\ \mathsf{ok}$ and $\Upsilon,\Theta' \vdash \mathsf{H},(\mathsf{l}{:}\langle \mathsf{y}\rangle)$. The rest follows from the substitution lemma.

B Generating Code with Arbitrary Many Variables

Our example is the Fibonacci-like function, described in [6, §2.4]:

```
let gib = fix (λf.λx.λy.λn.
  if n=0 then x else if n=1 then y else f y (x+y) (n−1))
```

For example, gib 1 1 5 returns 8. The naive specialization to the given n

```
let gib_naive =
  let body = fix (λf.λx.λy.λn.
  if n=0 then x else if n=1 then y else f y (x+y) (n−1))
  in λn.λx.λy. body x y n
```

is unsatisfactory: gib_naive 5 generates

```
λx.λy. (y + (x + y)) + ((x + y) + (y + (x + y)))
```

with many duplicates, exponentially degrading performance. A slight change

```
let gibs =
  let body : ∀ γ. ⟨int⟩^γ → ⟨int⟩^γ → int → ⟨int⟩^γ = fix (λf.λx.λy.λn.
  if n=0 then x else if n=1 then y else clet z = (x+y) in f y z (n−1))
  in λn.λx.λy. body x y n
```

gives a much better result: gibs 5 produces

```
λx.λy. (λz. (λu. (λw. (λx1.x1) (u + w)) (z + u)) (y + z)) (x + y)
```

which runs in linear time. The improved generator relies on polymorphic recursion: that is why the signature is needed.