Code Generation and Hybrid Logic

Shunsuke Yamasaki and Oleg Kiselyov

Tohoku University, Japan
oleg@okmij.org

Abstract. Although code generation (staged) calculi are developed to serve as models of program generation, they also turned out to have deep connections to various systems of modal logic.
A recent addition to the family of staged calculi is <NJ>, designed to model program generators that may use mutable cells to store and retrieve fragments of the generated code. Even though <NJ> permits storing fragments with not yet bound variables, <NJ>'s type system, modeled after region calculi, nevertheless ensures that the generated code is well-typed and well-scoped.
This paper proposes a logical interpretation of <NJ>. First, we distill the calculus to a simpler <$\underline{\texttt{NJ}}$>. We then introduce an intuitionistic hybrid logic $\mathcal{NJ}^{\gamma}$, describe its Kripke semantics, specify the natural deduction proof system and prove its soundness. We then relate the type system of <$\underline{\texttt{NJ}}$> to $\mathcal{NJ}^{\gamma}$.

1 Introduction

This work originates from the study of program generation, specifically, staging: programming language systems with facilities to generate code for later use. The long history of code generation (see [11] for an overview) taught the importance of static correctness guarantees about the generated code. At the very least, we ought to ascertain that the generated code be (i) well-formed; (ii) well-typed; and (iii) well-scoped. Well-scopedness is rather subtle [6, 10]; in the first approximation one may think of it as the absence of unbound variables in the generated (or, 'future-stage') code. The three requirements together ensure that the generated code shall compile without errors. The user would hence be spared from debugging or even looking at that code (which is often obscure to the point of unreadable).

With such guarantees in mind, there have been developed a number of calculi, starting from $\lambda^{\Box}$ [5] and λ° [4] (the latter underlying the Typed Template Haskell, among others) to λ^{α} [12] (influencing MetaOCaml) and their many successors. All these calculi stress ensuring the three guarantees *by construction*: not only the complete generated code should be correct (in the sense of the three criteria), but also the fragments still under construction. The insistence on the static correctness by construction is not a luxury: [9] reports from the real-life experience with a large-scale code generation system that detecting unbound variables only when compiling the complete generated code is too late. It proved truly time consuming to determine what part of a large generator has lead to one variable out of several thousands to be used before bound.

Although stage calculi have been developed from pragmatic motivations of code generation, they turn out to have deep logical connections: to the intuitionistic modal logic S4 (for $\lambda^{\Box}$), to a linear-time temporal logic and lax modality (λ°) to the contextual modal type theory [8]. We report on logical connections of the recently introduced staged calculus `<NJ>` [7] to a variant of intuitionistic hybrid logic.

The calculus `<NJ>` stands out in allowing effects during code generation, specifically, storing code fragments in mutable cells. Code-generation effects are indispensable for let- and assertion-insertion, loop-invariant code motion and loop interchange [6]. Storing potentially open code fragments in mutable cells also brings in the danger of moving a part of code beyond the scope of its bindings, resulting in a program with unbound or unexpectedly bound variables: so-called 'scope extrusion'. The calculus `<NJ>` statically prevents such problems. As many other stage calculi, `<NJ>` has been pragmatically motivated. The experience with other stage calculi suggests that it may be worth looking at `<NJ>` from a more theoretical point of view: Exactly which features of `<NJ>`'s type system are responsible for the prevention of scope extrusion? Can they be untied from the semantics of reference cells and understood in terms of some formal logic system? The type system of `<NJ>` has rather complicated rules; one may hope the logical point of view help clarify them.

As intimated already in [7] and fully explained in §2.2, the key feature of `<NJ>` is a rather odd-looking typing rule (CAbs), which, in turn, relies on annotating code types with the representation of binding environment. The annotation, by a symbol called refined environment classifier thus points out the environment in which the code type 'makes sense'. Nominals of the hybrid tense logic (hybrid logic in general) [2] immediately spring to mind.

Although the connection between `<NJ>` and hybrid logic is intuitively clear, making it precise turns out not straightforward. Most systems of hybrid logic are based on classical propositional logic. The rare exceptions [3] are intuitionizing classical hybrid modal logic (taking direct product of Kripke intuitionistic possible-world and hybrid modal logic semantics). For connection with `<NJ>` we want a hybridized intuitionistic logic – which we develop in this paper and use it to understand the type system of `<NJ>`.

Specifically, our contributions are:

– a subset $\texttt{<}\underline{\texttt{NJ}}\texttt{>}$ of the full `<NJ>` without the artifacts caused by the small-step dynamic semantics of `<NJ>` and without reference cells, but possessing all characteristics of `<NJ>` including modeling of scope extrusion;
– an intuitionistic hybrid logic $\mathcal{NJ}^{\gamma}$ with the standard Kripke possible-world semantics and the nominals that let logical formulas refer to a particular possible world;
– a natural deduction system for $\mathcal{NJ}^{\gamma}$, with a proof of its soundness (and hence consistency);

– an interpretation of <$\underline{\mathtt{NJ}}$> types as $\mathcal{NJ}^\gamma$ formulas and a proof that the typing rules of <$\underline{\mathtt{NJ}}$> are admissible. Thus <$\underline{\mathtt{NJ}}$> terms are proof witnesses of $\mathcal{NJ}^\gamma$ propositions expressed by their types.

The structure of the paper is as follows. First we remind of <NJ> and introduce its subset <$\underline{\mathtt{NJ}}$>. §3 introduces $\mathcal{NJ}^\gamma$: its semantics §3.1, natural deduction proof system §3.2 and the proof of its soundness. Next, §4 describes the interpretation of <$\underline{\mathtt{NJ}}$>'s types as $\mathcal{NJ}^\gamma$ formulas and proves the admissibility of <$\underline{\mathtt{NJ}}$>'s typing rules. We review the related work in §5.

2 The calculus <$\underline{\mathtt{NJ}}$>

This section reminds of <NJ> by the way of introducing its subset <$\underline{\mathtt{NJ}}$>, with only the features needed for our task of relating <NJ>'s type system with $\mathcal{NJ}^\gamma$.

<$\underline{\mathtt{NJ}}$>, Figure 1, is an extension of the simply-typed lambda-calculus with constants and expression forms that create or combine program code ('code values'). The (generated) program code is often called 'future-stage' because it is to be executed only in the future, when compiled and run. The code that can be executed now is, hence, 'present-stage'. Code values represent future-stage expressions that can be combined into bigger expressions (but not executed!) now. A future-stage expression of type t is represented as a code value of the type $\langle\mathsf{t}\rangle^\gamma$. Code value may contain free variables, which are described by the binding environment γ (so-called 'environment classifier') as discussed later. Beside code types, <$\underline{\mathtt{NJ}}$> has unspecified base types (for the sake of examples we assume they include integers and booleans), function types $\mathsf{t}_1{\to}\mathsf{t}_2$ and product types $\mathsf{t}_1 * \mathsf{t}_2$. (Sum types are elided for brevity; they are straightforward to add.) <NJ> was designed to be a two-stage language (capable of generating code but not code generators), therefore, in $\langle\mathsf{t}\rangle^\gamma$, t itself does not contain any code types.

Variables	$\mathsf{x,y,z,u,f,n,r}\ldots$
Classifier	γ
Base Types	b
Simple Types	$\mathsf{s} ::= \mathsf{b} \mid \mathsf{s}\to\mathsf{s} \mid \mathsf{s} * \mathsf{s}$
Types	$\mathsf{t} ::= \mathsf{b} \mid \mathsf{t}\to\mathsf{t} \mid \mathsf{t} * \mathsf{t} \mid \langle\mathsf{s}\rangle^\gamma$
Expressions	$\mathsf{e} ::= \mathsf{x} \mid \mathsf{c}_0 \mid \mathsf{c}_1\ \mathsf{e} \mid \mathsf{c}_2\ \mathsf{e}\ \mathsf{e} \mid \lambda\mathsf{x}.\ \mathsf{e} \mid \underline{\lambda}\mathsf{x}.\ \mathsf{e} \mid \mathsf{e}\ \mathsf{e}$

Fig. 1. Syntax of <$\underline{\mathtt{NJ}}$>

Expressions are the familiar variable references, lambda-abstractions and applications – and 'constant expressions', which are constants applied to the appropriate number of arguments according to their arity. The constants c_i with their arities i are defined in Fig. 2. In particular, values of pair types are created by the expression $\mathsf{pair}\ \mathsf{e}_1\ \mathsf{e}_2$. 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 constants and code combinators as needed. Although the constants may have function types, they are not expressions, unless applied to the right number of arguments.

Arity 0

$1,2,3,\ldots \;:\; \mathsf{int}$

$\textbf{true}, \textbf{false}\!: \mathsf{bool}$

Arity 1

$\underline{\mathsf{cint}}\!: \; \mathsf{int} \to \langle\mathsf{int}\rangle^{\gamma}$

$\underline{\mathsf{cbool}}\!: \mathsf{bool} \to \langle\mathsf{bool}\rangle^{\gamma}$

Arity 2

$\mathsf{pair}\!: \; \mathsf{t}_1 \to \mathsf{t}_2 \to \mathsf{t}_1 * \mathsf{t}_2$

$\underline{\mathsf{pair}}\!: \langle\mathsf{t}_1\rangle^{\gamma} \to \langle\mathsf{t}_2\rangle^{\gamma} \to \langle\mathsf{t}_1 * \mathsf{t}_2\rangle^{\gamma}$

$+\!: \; \mathsf{int} \to \mathsf{int} \to \mathsf{int}$

$\underline{+}\!: \; \langle\mathsf{int}\rangle^{\gamma} \to \langle\mathsf{int}\rangle^{\gamma} \to \langle\mathsf{int}\rangle^{\gamma}$

$\underline{@}\!: \; \langle\mathsf{t}_1 {\to} \mathsf{t}_2\rangle^{\gamma} \to \langle\mathsf{t}_1\rangle^{\gamma} \to \langle\mathsf{t}_2\rangle^{\gamma}$

Fig. 2. The constants c_i of <$\underline{\mathtt{NJ}}$> with their arities i

We describe the dynamic semantics of `<NJ>` only briefly, to give intuition for code generation. The formal presentation is unnecessary since <$\underline{\mathtt{NJ}}$>, the subject of study here, is the subset that deals with type-checking of expressions, not with their reductions. Whereas other stage calculi use dedicated expression forms for generating code (such as quotes in Lisp, brackets of λ^{α}), <$\underline{\mathtt{NJ}}$> uses code combinators. For example, the present-stage expression $\underline{\mathsf{cint}}\ 1$ generates the code with the literal 1; likewise, $\underline{\mathsf{cbool}}$ generates a literal boolean. More complex expressions are built by combining the already generated fragments. Whereas $1 + 2$ is a present-stage expression made of the constant $+$ (written in infix) applied to two arguments, which evaluates to 3, $\underline{\mathsf{cint}}\ 1 \mathbin{\underline{+}} \underline{\mathsf{cint}}\ 2$ evaluates to the code value that represents the program adding two and three. No addition is performed now. As another example, $\underline{\mathsf{pair}}\ (\underline{\mathsf{cbool}}\ \textbf{true})\ (\underline{\mathsf{cint}}\ 2 \mathbin{\underline{+}} \underline{\mathsf{cint}}\ 3)$ produces the code $\mathsf{pair}\ \textbf{true}\ (2{+}3)$.

Functions are generated by the $\underline{\lambda}\mathsf{x}.\ \mathsf{e}$ form (which is the only form that distinguishes <$\underline{\mathtt{NJ}}$> from the standard lambda-calculus): $\underline{\lambda}\mathsf{x}.\ \mathsf{x} \mathbin{\underline{+}} (\underline{\mathsf{cint}}\ 1)$ produces the code $\lambda\mathsf{y}.\ \mathsf{y} + 1$, where y is a fresh name. First, a fresh name, say, y, for the future-stage variable is generated and substituted for x; then the code for the body is built; finally, the binder $\lambda\mathsf{y}$ for the so-far free y is generated. Unlike the ordinary $\lambda\mathsf{x}$, the body of $\underline{\lambda}\mathsf{x}$ *is* evaluated.

<$\underline{\mathtt{NJ}}$> is (intentionally) a mere shadow of `<NJ>`: the latter also has fixpoints, lists, and, specifically, references. Nevertheless, <$\underline{\mathtt{NJ}}$> reproduces characteristic features of `<NJ>`, including the prevention of scope extrusion, as we explain in §2.2.

2.1 Type System

The typing judgement $\Gamma \vdash \mathsf{e}\!: \mathsf{t}$ states that the expression e has the type t in the typing environment Γ. The environment

$$\Gamma \;::=\; [] \mid \Gamma, \gamma \mid \Gamma, (\gamma_1 {\succ} \gamma_2) \mid \Gamma, \mathsf{x}{:}\mathsf{t}$$

is an ordered sequence associating types with free variables in an expression. Γ also contains classifiers γ and classifier subtyping witnesses $\gamma_1 {\succ} \gamma_2$ to be explained shortly. We will always assume that Γ is well-formed: all free variables, classifiers and subtyping witnesses are unique; any classifier that appears in Γ as part of a subtyping witness or the type must have appeared as an element earlier (that is, classifiers must be defined before use). We write Γ,Γ' and for the concatenation of two sequences such that the result must be well-formed. Intuitively, classifiers are references to binding environments. The initial environment Γ contains only γ_0, representing the empty binding environment.

$$\frac{}{\Gamma \vdash \mathsf{c}\colon \mathsf{tc}}\,\text{Const} \qquad \frac{\mathsf{x}{:}\mathsf{t} \in \Gamma}{\Gamma \vdash \mathsf{x}\colon \mathsf{t}}\,\text{Var} \qquad \frac{\Gamma \vdash \mathsf{e}\colon \langle \mathsf{t}\rangle^{\gamma_1} \qquad \Gamma \models \gamma_2 \succ \gamma_1}{\Gamma \vdash \mathsf{e}\colon \langle \mathsf{t}\rangle^{\gamma_2}}\,\text{Sub}$$

$$\frac{\Gamma \vdash \mathsf{e}_1\colon \mathsf{t}_1 \to \mathsf{t}_2 \quad \Gamma \vdash \mathsf{e}_2\colon \mathsf{t}_1}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2\colon \mathsf{t}_2}\,\text{App}$$

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

Fig. 3. Type system

Thus to type-check $\underline{\lambda}\mathsf{x}.\mathsf{e}$, e should have the type annotated with a fresh classifier γ_1 in the context extended with $\mathsf{x}{:}\langle \mathsf{t}_1\rangle^{\gamma_1}$ and the subtyping witness that lets e use the code values marked with the old γ. The new γ_1 hence represents the environment that is an extension of γ's. 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 rule (Sub) relies on the partial order on classifiers: $\Gamma \models \gamma_2 \succ \gamma_1$ if either $\gamma_2 \succ \gamma_1$ literally occurs in Γ as a witness, or can be derived by reflexivity and transitivity.

As an example, the following is a typing derivation for the generator of the $\mathsf{t}_1 \to \mathsf{t}_2 \to \mathsf{t}_1 * \mathsf{t}_2$ function. The derivation illustrates the use of the Sub rule. We have assumed Γ_1 to be $\gamma_0,\gamma_1,\gamma_1 \succ \gamma_0,\ \mathsf{x}_1{:}\langle \mathsf{t}_1\rangle^{\gamma_1},\gamma_2,\gamma_2 \succ \gamma_1,\mathsf{x}_2{:}\langle \mathsf{t}_2\rangle^{\gamma_2}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \underline{\mathsf{pair}}\colon \langle \mathsf{t}_1\rangle^{\gamma_2} \to \langle \mathsf{t}_2\rangle^{\gamma_2} \to \langle \mathsf{t}_1 * \mathsf{t}_2\rangle^{\gamma_2}} \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash \mathsf{x}_1\colon \langle \mathsf{t}_1\rangle^{\gamma_1}} \quad \Gamma_1 \models \gamma_1 \succ \gamma_2}{\Gamma_1 \vdash \mathsf{x}_1\colon \langle \mathsf{t}_2\rangle^{\gamma_2}}\,\text{Sub} \qquad \dfrac{}{\Gamma_1 \vdash \mathsf{x}_2\colon \langle \mathsf{t}_2\rangle^{\gamma_2}}}{\Gamma_1 \vdash \underline{\mathsf{pair}}\ \mathsf{x}_1\ \mathsf{x}_2\colon \langle \mathsf{t}_1 * \mathsf{t}_2\rangle^{\gamma_2}}\,\text{App}}{\gamma_0,\gamma_1,\gamma_1 \succ \gamma_0,\mathsf{x}{:}\langle \mathsf{t}_1\rangle^{\gamma_1} \vdash \underline{\lambda}\mathsf{x}_2.\ \underline{\mathsf{pair}}\ \mathsf{x}_1\ \mathsf{x}_2\colon \langle \mathsf{t}_2 \to \mathsf{t}_1 * \mathsf{t}_2\rangle^{\gamma_1}}\,\text{CAbs}}{\gamma_0 \vdash \underline{\lambda}\mathsf{x}_1.\ \underline{\lambda}\mathsf{x}_2.\ \underline{\mathsf{pair}}\ \mathsf{x}_1\ \mathsf{x}_2\colon \langle \mathsf{t}_1 \to \mathsf{t}_2 \to \mathsf{t}_1 * \mathsf{t}_2\rangle^{\gamma_0}}\,\text{CAbs}$$

2.2 Scope Extrusion

`<NJ>` is designed to model code generation with effects, in particular, mutable cells. Storing generated code in mutable cells brings in the danger of scope extrusion, or the 'leaking' of a future-stage variable beyond the scope of its binder. To show the problem, we temporarily add to `<NJ>` the type of reference cells $\mathsf{t}\ \textbf{ref}$ and the corresponding constants

$\textbf{ref} : \ \mathsf{t} \to \mathsf{t}\ \textbf{ref} \qquad ! : \mathsf{t}\ \textbf{ref} \to \mathsf{t} \qquad := : \mathsf{t}\ \textbf{ref} \to \mathsf{t} \to \mathsf{t}$

(as we shall soon see, the scope extrusion problem can be understood without any reference cells).

Consider the following code:

```
let r = ref (cint 3) in
let z = λx. r:= x in
!r
```

(where **let** x = e$_1$ **in** e$_2$ is syntax sugar for (λx. e$_2$) e$_1$). Recall, $\underline{\lambda}$x. r:= x is evaluated by first generating a unique name for a future-stage variable and substituting it for x. The later evaluated body stores that future-stage variable in an outside reference cell, from which it is extracted at the end. The end result is a program made of an unbound variable.

Let us examine what exactly makes <NJ> with reference cells (and, hence, <NJ>) reject this code. (The explanation here is a significantly expanded version of the explanation in [7, §3.1].) First, consider the safe version of the code

```
let r = ref (cint 3) in λx. pair !r x
```

which is accepted by the type checker. Here is the derivation (we write Γ for γ_0, r:$\langle\mathsf{int}\rangle^{\gamma_0}$ **ref**, γ_1, $\gamma_1 \succ \gamma_0$, x:$\langle\mathsf{t}\rangle^{\gamma_1}$):

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \ !\mathsf{r}\colon \langle\mathsf{int}\rangle^{\gamma_0} \qquad \Gamma \models \gamma_1 \succ \gamma_0}{\Gamma \vdash \ !\mathsf{r}\colon \langle\mathsf{int}\rangle^{\gamma_1}}\,\text{Sub} \qquad \Gamma \vdash \mathsf{x}\colon\langle\mathsf{int}\rangle^{\gamma_1}}{\gamma_0,\ \mathsf{r}\colon\langle\mathsf{int}\rangle^{\gamma_0}\ \textbf{ref},\ \gamma_1,\ \gamma_1\succ\gamma_0,\ \mathsf{x}\colon\langle\mathsf{t}\rangle^{\gamma_1} \vdash \underline{\mathsf{pair}}\ !\mathsf{r}\ \mathsf{x}\colon \langle\mathsf{int*t}\rangle^{\gamma_1}}}{\gamma_0 \vdash \textbf{let}\ \mathsf{r}{=}\textbf{ref}\ (\underline{\mathsf{cint}}\ 3)\ \textbf{in}\ \underline{\lambda}\mathsf{x}.\ \underline{\mathsf{pair}}\ !\mathsf{r}\ \mathsf{x}\colon \langle\mathsf{t}\to\mathsf{int*t}\rangle^{\gamma_0}}$$

Recall, $\underline{\mathsf{pair}}$ (see Fig.2) applies only to the code values annotated with the same classifier. However, the reference cell r holds the code fragment annotated with the classifier γ_0 (because γ_1 is not available outside the $\underline{\lambda}$x expression) but x is annotated with γ_1. Fortunately, we can apply (Sub) to the contents extracted from r, taking advantage of the presence $\gamma_1 \succ \gamma_0$ in Γ.

Now consider the scope extrusion code. We attempt the typing derivation as

$$\dfrac{\dfrac{\textit{stuck}!}{\gamma_0,\ \mathsf{r}\colon\langle\mathsf{int}\rangle^{\gamma_0}\ \textbf{ref},\ \gamma_1,\ \gamma_1\succ\gamma_0,\ \mathsf{x}\colon\langle\mathsf{int}\rangle^{\gamma_1} \vdash \mathsf{r} := \mathsf{x}\colon \langle\mathsf{int}\rangle^{\gamma_1}}}{\gamma_0 \vdash \textbf{let}\ \mathsf{r}{=}\textbf{ref}\ (\underline{\mathsf{cint}}\ 3)\ \textbf{in}\ \underline{\lambda}\mathsf{x}.\ \mathsf{r} := \mathsf{x}\colon \langle\mathsf{int}\to\mathsf{int}\rangle^{\gamma_0}}$$

And here we are stuck: r has the type $\langle\mathsf{int}\rangle^{\gamma_0}$ **ref** with the classifier γ_0 but x:$\langle\mathsf{int}\rangle^{\gamma_1}$ is annotated with γ_1. A reference cell may be updated only with values of the single type. We cannot give r the type $\langle\mathsf{int}\rangle^{\gamma_1}$ **ref** to start with, because γ_1 is not available outside the $\underline{\lambda}$x expression; we cannot use (Sub) to derive x:$\langle\mathsf{int}\rangle^{\gamma_0}$ because the needed $\gamma_0 \succ \gamma_1$ does not hold (in fact, the opposite is true). Finally, we cannot use subtyping to convert the type of r to $\langle\mathsf{int}\rangle^{\gamma_1}$ **ref**, because the **ref** type is non-variant.

Thus the reason the scope extrusion code is rejected by the type-checker is (i) a fresh classifier γ_1 cannot leak from its subderivation (akin to the eigenvariable condition in natural deduction); and (ii) non-variance of reference cell types. The (CAbs) rule was responsible for the former property, by introducing the fresh γ_1 in the first place. We can model and understand the typing problem with the scope extrusion code without any reference cells: after all, function types like $\mathsf{t}\to\mathsf{t}$ are also non-variant.

First, we consider the following valid derivation

$$\dfrac{\begin{array}{c}\vdots\\ \gamma_0,\ \mathsf{u}{:}\langle\mathsf{int}\rangle^{\gamma_0},\ \gamma_1,\ \gamma_1\succ\gamma_0,\ \mathsf{x}{:}\langle\mathsf{t}\rangle^{\gamma_1} \vdash \underline{\mathsf{pair}}\ \mathsf{u}\ \mathsf{x}{:}\ \langle\mathsf{int}*\mathsf{t}\rangle^{\gamma_1}\end{array}}{\gamma_0 \vdash \mathbf{let}\ \mathsf{u}{=}\underline{\mathsf{cint}}\ 1\ \mathbf{in}\ \underline{\lambda}\mathsf{x}.\ \underline{\mathsf{pair}}\ \mathsf{u}\ \mathsf{x}{:}\ \langle\mathsf{t}\to\mathsf{int}*\mathsf{t}\rangle^{\gamma_0}}$$

Although $\mathsf{u}{:}\langle\mathsf{int}\rangle^{\gamma_0}$ and $\mathsf{x}{:}\langle\mathsf{t}\rangle^{\gamma_1}$, with different classifiers, the expression $\underline{\mathsf{pair}}\ \mathsf{u}\ \mathsf{x}$ is well-typed because the (Sub) rule lets us derive $\mathsf{u}{:}\langle\mathsf{int}\rangle^{\gamma_1}$. However, a rather similar

$$\dfrac{\dfrac{\textit{stuck}!}{\gamma_0,\ \mathsf{f}{:}\langle\mathsf{t}\rangle^{\gamma_0}\to\langle\mathsf{int}*\mathsf{t}\rangle^{\gamma_0},\ \gamma_1,\ \gamma_1\succ\gamma_0,\ \mathsf{x}{:}\langle\mathsf{t}\rangle^{\gamma_1} \vdash \mathsf{f}\ \mathsf{x}{:}\ \langle\mathsf{int}*\mathsf{t}\rangle^{\gamma_1}}\,\text{App}}{\gamma_0 \vdash \mathbf{let}\ \mathsf{f}{=}\lambda\mathsf{z}.\ \underline{\mathsf{pair}}\ (\underline{\mathsf{cint}}\ 1)\ \mathsf{z}\ \mathbf{in}\ \underline{\lambda}\mathsf{x}.\ \mathsf{f}\ \mathsf{x}{:}\ \langle\mathsf{t}\to\mathsf{int}*\mathsf{t}\rangle^{\gamma_0}}$$

is stuck: to apply (App), the types of x and of the f's argument should have the same classifiers, but they do not, and (Sub) cannot be used to make them the same. The only other way to complete the derivation is to give f the type $\langle\mathsf{t}\rangle^{\gamma_1} \to \langle\mathsf{int}*\mathsf{t}\rangle^{\gamma_1}$ – which leaks the local γ_1 to a wider scope.

Granted, if we are to run the program just rejected, no scope extrusion occurs. By the same token, no ill result comes from executing 1 + **if true then** 2 **else false**, which is nevertheless ill-typed in many type systems. <NJ> hence rejects programs which the closely related λ° would have accepted. However, the latter cannot be safely extended with reference cells but the former can.

Once we have understood how the dynamic problem of scope extrusion is statically prevented, we can dispense with reference cells in <NJ>, keeping it small and normalizing. After all, the crucial rule (CAbs) does not involve them. Our goal is to study that rule logically.

3 Intuitionistic Hybrid Logic $\mathcal{NJ}^\gamma$

Hybrid logic arose from the work of Arthur N. Prior on tense logic, to give meaning to the sentences like "It is 1am on January 7, 2018", which are true at only one specific moment. He introduced 'nominals' – special kind of propositional letters; a nominal is true only at exactly one possible world, and, hence, in effect 'names' that world. Nominals, therefore, let logical formulas refer to possible worlds. This section develops an intuitionistic hybrid logic $\mathcal{NJ}^\gamma$, with its Kripke semantics and natural deduction proof theory. Its connection with <NJ> is described in §4 (and its relation with other hybrid and modal logics, in §5).

$\mathcal{NJ}^\gamma$ is a propositional logic, whose formulas are:

Formulas	$\mathsf{A},\mathsf{B} ::= \mathsf{p} \mid \gamma \mid \mathsf{A} \wedge \mathsf{A} \mid \mathsf{A} \to \mathsf{A} \mid \Box\mathsf{A} \mid \Diamond\mathsf{A} \mid @_\gamma \mathsf{A}$
Simple Formulas	Formulas without nominals
Set of formulas	Γ

where p is a propositional letter and γ is a nominal (a special kind of propositional letters). For the sake of connection with <NJ> we do not include $\bot$ and negation: our $\mathcal{NJ}^\gamma$ is hence minimal. We also omit disjunction – but this is for brevity (for the same reason, <NJ> elides sums). Both are straightforward to add. $\mathcal{NJ}^\gamma$ also has so-called 'satisfaction statements' $@_\gamma \mathsf{A}$, which are meant to assert that A is true in the world named by γ. For example, if worlds describe time instances and γ is true in the world corresponding to 1am, January 7, 2018 and p is the proposition "it is snowing", then the formula $@_\gamma \mathsf{p}$ represents the meaning of the sentence "It is snowing at 1am on January 7, 2018".

We write $\mathsf{NOM}(\mathsf{A})$ and $\mathsf{NOM}(\Gamma)$ for the set of nominals occurring in a formula (resp. set of formulas): $\mathsf{NOM}(\gamma)$ is $\{\gamma\}$, $\mathsf{NOM}(@_\gamma \mathsf{A}) = \mathsf{NOM}(\mathsf{A}) \cup \{\gamma\}$, with the other cases being homomorphisms. Simple formulas are formulas without nominals as a component (but, possibly, with satisfaction statements).

3.1 Kripke semantics

We now define the Kripke possible-world semantics for $\mathcal{NJ}^\gamma$, which is the semantics for intuitionistic logic with added nominals and satisfaction statements.

A model $\mathcal{M}$ is a tuple $(\mathsf{W},\leq,\mathsf{g},\mathsf{V})$ where

- W is a non-empty set whose elements, denoted v or w, are called worlds;
- $\leq$ is a partial order on W (i.e., reflexive, anti-symmetric and transitive relation)
- g, called assignment, is a surjective function from nominals to worlds: for each nominal it tells the world in which it is true.
- V is a function on W such that for each $\mathsf{w} \in \mathsf{W}$ gives a set of propositional letters. V has to be monotone: if $\mathsf{w} \leq \mathsf{v}$ then $\mathsf{V}(\mathsf{w}) \subset \mathsf{V}(\mathsf{v})$.

A frame is $(\mathsf{W},\leq)$, i.e., model without valuations.

The truth of a formula A in a model $\mathcal{M} = (\mathsf{W},\leq,\mathsf{g},\mathsf{V})$ at a world $\mathsf{w} \in \mathsf{W}$ – written as $\mathcal{M},\mathsf{w} \models \mathsf{A}$ – is inductively defined as follows:

$\mathcal{M},\mathsf{w} \models \mathsf{p}$	iff	$\mathsf{p} \in \mathsf{V}(\mathsf{w})$
$\mathcal{M},\mathsf{w} \models \gamma$	iff	$\mathsf{g}(\gamma) = \mathsf{w}$
$\mathcal{M},\mathsf{w} \models \mathsf{A} \wedge \mathsf{B}$	iff	$\mathcal{M},\mathsf{w} \models \mathsf{A}$ and $\mathcal{M},\mathsf{w} \models \mathsf{B}$
$\mathcal{M},\mathsf{w} \models \mathsf{A} \to \mathsf{B}$	iff	for all v such that $\mathsf{w}\leq\mathsf{v}$, $\mathcal{M},\mathsf{v} \models \mathsf{A}$ implies $\mathcal{M},\mathsf{v} \models \mathsf{B}$
$\mathcal{M},\mathsf{w} \models \Box\mathsf{A}$	iff	for all v such that $\mathsf{w}\leq\mathsf{v}$, $\mathcal{M},\mathsf{v} \models \mathsf{A}$
$\mathcal{M},\mathsf{w} \models \Diamond\mathsf{A}$	iff	there exists v such that $\mathsf{w}\leq\mathsf{v}$ and $\mathcal{M},\mathsf{v} \models \mathsf{A}$
$\mathcal{M},\mathsf{w} \models @_\gamma \mathsf{A}$	iff	$\mathcal{M},\mathsf{g}(\gamma) \models \mathsf{A}$

The relation $\models$ straightforwardly extends to a set of formulas. As common, we say A is *valid in the model* $\mathcal{M}$, written as $\mathcal{M} \models \mathsf{A}$, iff it is valid at every world w in the model. A formula A is *valid*, (written $\models \mathsf{A}$) just in case it is valid in every model. Finally, we write $\Gamma \models \mathsf{A}$ whenever $\mathcal{M} \models \Gamma$ implies $\mathcal{M} \models \mathsf{A}$ for all models $\mathcal{M}$.

Our relation $\models$ is rather standard for hybrid logics (see, for example, [1, 2]). In particular, from the truth of $\mathcal{M},\mathsf{g}(\gamma) \models \mathsf{A}$ follows the truth of $\mathcal{M},\mathsf{w} \models @_\gamma\ \mathsf{A}$ for any world w whatsoever – which *is* the intended meaning of the satisfaction operator, see [1, 2]. What is specific for our $\models$ relation is the meaning we give to $\mathsf{A}\rightarrow\mathsf{B}$, motivated by the (CAbs) and (Sub) rules of `<NJ>`.

One can easily verify the following (we write $@_\gamma\ \Gamma$ for a set of formulas obtained by attaching $@_\gamma$ to each formula in Γ):

Proposition 1. *$\Gamma \models A$ iff $@_\gamma\ \Gamma \models @_\gamma A$ for any $\gamma \notin NOM(\Gamma) \cup NOM(A)$.*

Proposition 2 (Monotonicity of simple formula). *For any simple formula B, if $\mathcal{M},w \models B$ and $w \leq v$ then $\mathcal{M},v \models B$.*

3.2 Natural Deduction

Figure 4 presents the proof system for $\mathcal{NJ}^\gamma$, in natural deduction style, deriving $\Gamma \vdash \mathsf{A}$ for a formula A and a set of its assumptions Γ.

$$\dfrac{\mathsf{A} \in \Gamma}{\Gamma \vdash \mathsf{A}}\,\text{Assm} \qquad \dfrac{}{\Gamma \vdash @_\gamma\ \gamma}\,\text{Refl1} \qquad \dfrac{}{\Gamma \vdash @_\gamma \Diamond \gamma}\,\text{Refl2} \qquad \dfrac{\Gamma \vdash @_\gamma\ \mathsf{A} \quad \Gamma \vdash @_\gamma\ \gamma_1}{\Gamma \vdash @_{\gamma_1}\ \mathsf{A}}\,\text{EqR}$$

$$\dfrac{\Gamma,\ @_\gamma\ \mathsf{A} \vdash \mathsf{B} \quad \Gamma \vdash @_\gamma\ \gamma_1}{\Gamma,\ @_{\gamma_1}\ \mathsf{A} \vdash \mathsf{B}}\,\text{EqL} \qquad \dfrac{\Gamma \vdash @_\gamma \Diamond \gamma_1 \quad \Gamma \vdash @_{\gamma_1} \Diamond \gamma_2}{\Gamma \vdash @_\gamma \Diamond \gamma_2}\,\text{Trans}$$

$$\dfrac{\Gamma \vdash @_\gamma\ \mathsf{A}}{\Gamma \vdash @_{\gamma'}\ @_\gamma\ \mathsf{A}}\,@\text{I} \qquad \dfrac{\Gamma \vdash @_{\gamma'}\ @_\gamma\ \mathsf{A}}{\Gamma \vdash @_\gamma\ \mathsf{A}}\,@\text{E1} \qquad \dfrac{@_{\gamma'}\ \Gamma \vdash @_{\gamma'}\ \mathsf{A}}{\Gamma \vdash \mathsf{A}}\,@\text{E2}^*$$

$$\dfrac{\Gamma \vdash @_\gamma\ \mathsf{A} \quad \Gamma \vdash @_\gamma\ \mathsf{B}}{\Gamma \vdash @_\gamma\ (\mathsf{A} \wedge \mathsf{B})}\,\wedge\text{I} \qquad \dfrac{\Gamma \vdash @_\gamma\ (\mathsf{A} \wedge \mathsf{B})}{\Gamma \vdash @_\gamma\ \mathsf{A}}\,\wedge\text{E1} \qquad \dfrac{\Gamma \vdash @_\gamma\ (\mathsf{A} \wedge \mathsf{B})}{\Gamma \vdash @_\gamma\ \mathsf{B}}\,\wedge\text{E2}$$

$$\dfrac{\Gamma,\ @_\gamma \Diamond \gamma\prime,\ @_{\gamma\prime}\ \mathsf{A} \vdash @_{\gamma\prime}\ \mathsf{B}}{\Gamma \vdash @_\gamma\ (\mathsf{A} \rightarrow \mathsf{B})}\,\rightarrow\text{I}^\star \qquad \dfrac{\Gamma \vdash @_\gamma\ (\mathsf{A} \rightarrow \mathsf{B}) \quad \Gamma \vdash @_\gamma\ \mathsf{A}}{\Gamma \vdash @_\gamma\ \mathsf{B}}\,\rightarrow\text{E}$$

$$\dfrac{\Gamma \vdash @_\gamma\ \mathsf{B}}{\Gamma \vdash @_\gamma \Box \mathsf{B}}\,\Box\text{I1}^\dagger \qquad \dfrac{\Gamma,\ @_\gamma \Diamond \gamma\prime \vdash @_{\gamma\prime}\ \mathsf{A}}{\Gamma \vdash @_\gamma \Box \mathsf{A}}\,\Box\text{I2}^* \qquad \dfrac{\Gamma \vdash @_\gamma \Diamond \gamma\prime \quad \Gamma \vdash @_\gamma \Box \mathsf{A}}{\Gamma \vdash @_{\gamma\prime}\ \mathsf{A}}\,\Box\text{E}$$

$*$ $\gamma' \notin \mathsf{NOM}(\Gamma) \cup \mathsf{NOM}(\mathsf{A})$

$\star$ $\gamma' \notin \mathsf{NOM}(\Gamma) \cup \mathsf{NOM}(\mathsf{A}) \cup \mathsf{NOM}(\mathsf{B}) \cup \{\gamma\}$

$\dagger$ B is a simple formula

Fig. 4. Natural deduction rules

Most of the rules are standard (see [3]). Some rules, however, stand out, most notably $\rightarrow$I. The rule $\Box$I1 is essentially the necessitation rule of modal logic, but it is asserted only for simple formulas (without nominals) ($@_\gamma\ \gamma$, which is true, clearly does not entail $@_\gamma\ \Box\gamma$, which can be false). This rule expresses the monotonicity, Thm. 2. Its side condition – B be a simple formula – makes the rule 'non-orthodox' (non-orthodox rules, with syntactic side conditions, frequently occur in proof systems of hybrid logic [1]).

The rules Refl2 and Trans are 'geometric' in that they express the structure of frames (the accessibility relation being reflexive and transitive).

The proof system lets us establish the following theorems

Theorem 1. *1.* $\Box A \vdash A$
2. $\Box A, \Box B \vdash \Box(A \wedge B)$
3. $\Box(A \to B), \Box A \vdash \Box B$
4. $\vdash A \to B \to A \wedge B$ *for simple formulas* A *and* B

Theorem 1-1 is what is called the M axiom of modal logic, and Theorem 1-3 is essentially the distribution rule (the defining rule of all normal logics, starting from K). Theorem 1-4 is derived as follows, where Γ is $@_{\gamma}\ \Diamond\gamma_1,\ @_{\gamma_1}\ \mathsf{A},\ @_{\gamma_1}\ \Diamond\gamma_2,\ @_{\gamma_2}\ \mathsf{B}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash @_{\gamma_1} \Diamond \gamma_2}\,\text{Assm} \qquad \dfrac{\dfrac{}{\Gamma \vdash @_{\gamma_1}\ \mathsf{A}}\,\text{Assm}}{\Gamma \vdash @_{\gamma_1} \Box\ \mathsf{A}}\,\Box\text{I1}}{\Gamma \vdash @_{\gamma_2}\ \mathsf{A}}\,\Box\text{E} \qquad \dfrac{}{\Gamma \vdash @_{\gamma_2}\ \mathsf{B}}\,\text{Assm}}{\Gamma \vdash @_{\gamma_2}\ (\mathsf{A} \wedge \mathsf{B})}\,\wedge\text{I}}{@_{\gamma}\ \Diamond\gamma_1,\ @_{\gamma_1}\ \mathsf{A} \vdash @_{\gamma_1}\ (\mathsf{B} \to \mathsf{A} \wedge \mathsf{B})}\,\to\text{I}}{\vdash @_{\gamma}\ (\mathsf{A} \to (\mathsf{B} \to \mathsf{A} \wedge \mathsf{B}))}\,\to\text{I}$$

The proof illustrates that even the →I rule is rather non-standard, we can still prove the expected theorems involving implication.

The proof system is sound with respect to the semantics in §3.1 (which also shows its consistency).

Theorem 2 (Soundness). *If* $\Gamma \vdash A$ *then* $\Gamma \models A$

Proof. The proof is by induction on the derivation of $\Gamma \vdash \mathsf{A}$. We show only a representative case of →I. That is, given $\Gamma \vdash @_{\gamma}(\mathsf{A} \to \mathsf{B})$ show that $\Gamma \models @_{\gamma}(\mathsf{A} \to \mathsf{B})$. The inductive hypothesis gives $\Gamma,\ @_{\gamma}\ \Diamond\gamma',\ @_{\gamma'}\ \mathsf{A} \models @_{\gamma'}\ \mathsf{B}$ where γ' is fresh. The latter condition lets us generalize $\forall\ \gamma'.\ \Gamma,\ @_{\gamma}\ \Diamond\gamma',\ @_{\gamma'}\ \mathsf{A} \models @_{\gamma'}\ \mathsf{B}$. Let $\mathcal{M}$ be a model, w' be its arbitrary world and $\mathsf{g}(\gamma)$ be w. Let v be an arbitrary world. From the surjectivity of g, let γ' be such that $\mathsf{g}(\gamma')$ is v. The inductive hypothesis, instantiated for w', and thus chosen γ' reads: $\mathcal{M},\mathsf{w'} \models \Gamma$, $\mathcal{M},\mathsf{w'} \models @_{\gamma}\ \Diamond\gamma'$, $\mathcal{M},\mathsf{w'} \models @_{\gamma'}\ \mathsf{A}$ implies $\mathcal{M},\mathsf{w'} \models @_{\gamma'}\ \mathsf{B}$. From the definition of the satisfaction statements, it can be re-written as $\mathcal{M},\mathsf{w'} \models \Gamma$, $\mathcal{M},\mathsf{w} \models \Diamond\gamma'$, $\mathcal{M},\mathsf{v} \models \mathsf{A}$ implies $\mathcal{M},\mathsf{v} \models \mathsf{B}$. Furthermore, $\mathcal{M},\mathsf{w} \models \Diamond\gamma'$ says that there exists the world w1 such that $\mathsf{w} \leq \mathsf{w1}$ and $\mathcal{M},\mathsf{w1} \models \gamma'$, which implies that w1 is v. Overall, we have $\mathcal{M},\mathsf{w'} \models \Gamma$, $\mathsf{w} \leq \mathsf{v}$, $\mathcal{M},\mathsf{v} \models \mathsf{A}$ implies $\mathcal{M},\mathsf{v} \models \mathsf{B}$ for an arbitrary v. This gives us $\mathcal{M},\mathsf{w} \models \mathsf{A} \to \mathsf{B}$, or $\mathcal{M},\mathsf{w'} \models @_{\gamma}\ (\mathsf{A} \to \mathsf{B})$.

We leave investigating the converse (the completeness) to the future work: as we shall see soon, the proof system is 'complete enough' for our main purpose of relating it to the type system of <NJ>.

4 <NJ> and $\mathcal{NJ}^\gamma$

This section introduces an interpretation of <NJ> types as $\mathcal{NJ}^\gamma$ formulas, and checks that the typing rules of <NJ> are all admissible in $\mathcal{NJ}^\gamma$. In short, we take the environment classifiers to be nominals and regard code types $\langle \mathsf{t} \rangle^\gamma$ of <NJ> as satisfaction statements $@_\gamma$ t in $\mathcal{NJ}^\gamma$.

First we introduce a particular class of frames (see §3.1), to be called *stage frames*:

- An isolated world w_B, accessible only to itself: $\mathsf{w}_B \leq \mathsf{v}$ implies that v is w_B itself;
- A world w_0 such that $\mathsf{w}_0 \leq \mathsf{v}$ for any other world v except w_B

In other words, the accessibility relation in stage frames is a partial order that can be represented as a tree rooted at w_0, plus the isolated point w_B. Likewise, we distinguish two particular nominals, γ_B and γ_0, such that $\mathsf{g}(\gamma_B) = \mathsf{w}_B$ and $\mathsf{g}(\gamma_0) = \mathsf{w}_0$ in any assignment g.

The proof rules in Figure 4 are amended with those in Fig. 5 which are sound

$$\frac{\Gamma \vdash @_{\gamma_B} \Diamond \gamma}{\Gamma \vdash @_{\gamma_B} \gamma}\,\text{Nom}$$

Fig. 5. Natural deduction rule for nominal

only for stage frames and represent the particular structure of stage frames.

4.1 <NJ>'s types and $\mathcal{NJ}^\gamma$'s formulas

All types of <NJ> are interpreted as satisfaction statements in $\mathcal{NJ}^\gamma$: code types $\langle \mathsf{t} \rangle^\gamma$ are interpreted as $@_\gamma$ t; all other types t are regarded as $@_{\gamma_B}$ t. Intuitively, the isolated w_B (and the corresponding nominal γ_B) represent the present-stage. The other worlds/nominals denote future-stage code; in particular, w_0 (and the nominal γ_0) marks the closed future-stage code and all the worlds represent open code with the various number of free variables. The relation $\mathsf{w} \leq \mathsf{v}$ (where w is not w_B) means that the future-stage world v contains all the free variables of w (and may add some). Thus the world accessibility relation corresponds to future-stage binding environment inclusion. In proof terms, this is written as $@_\gamma \Diamond\gamma_1$: the future-stage code classified by γ_1 has all free variables of the code classified by γ. It then follows that the <NJ>'s judgement $\Gamma \models \gamma_2 \succ \gamma_1$ is to be interpreted as $\Gamma \vdash @_{\gamma_1} \Diamond\gamma_2$.

4.2 Admissibility of <NJ>'s typing rules

We now consider that each typing rule of Fig. 3 is admissible in $\mathcal{NJ}^\gamma$.

$$\frac{}{\mathit{\Gamma} \vdash \mathsf{pair}\colon @_{\gamma_B}\ (\mathsf{A}\to\mathsf{B}\to\mathsf{A}\wedge\mathsf{B})}\ \text{ConstPair}$$

$$\frac{}{\mathit{\Gamma} \vdash \underline{\mathsf{pair}}\colon @_{\gamma_B}\ (@_{\gamma}\ \mathsf{A} \to @_{\gamma}\ \mathsf{B} \to @_{\gamma}\ \mathsf{A}\wedge\mathsf{B})}\ \text{ConstCPair}$$

$$\frac{}{\mathit{\Gamma} \vdash \underline{@}\colon @_{\gamma_B}\ (@_{\gamma}\ (\mathsf{A}\to\mathsf{B}) \to @_{\gamma}\ \mathsf{A} \to @_{\gamma}\ \mathsf{B})}\ \text{ConstCApp} \qquad \frac{\mathsf{x{:}A} \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash \mathsf{x}\colon \mathsf{A}}\ \text{Var}$$

$$\frac{\mathit{\Gamma} \vdash \mathsf{e}\colon @_{\gamma_1}\ \mathsf{A} \qquad \mathit{\Gamma} \vdash @_{\gamma_1}\ \Diamond\gamma_2}{\mathit{\Gamma} \vdash \mathsf{e}\colon @_{\gamma_2}\ \mathsf{A}}\ \text{Sub} \qquad \frac{\mathit{\Gamma} \vdash \mathsf{e}_1\colon @_{\gamma_B}\ (\mathsf{A}\to\mathsf{B}) \quad \mathit{\Gamma} \vdash \mathsf{e}_2\colon @_{\gamma_B}\ \mathsf{A}}{\mathit{\Gamma} \vdash \mathsf{e}_1\ \mathsf{e}_2\colon @_{\gamma_B}\ \mathsf{B}}\ \text{App}$$

$$\frac{\mathit{\Gamma},\ \mathsf{x}{:}@_{\gamma_B}\ \mathsf{A} \vdash \mathsf{e}\colon @_{\gamma_B}\ \mathsf{B}}{\mathit{\Gamma} \vdash \lambda\mathsf{x.e}\colon @_{\gamma_B}\ (\mathsf{A}\to\mathsf{B})}\ \text{Abs}$$

$$\frac{\gamma_1 \notin \mathsf{NOM}(\mathit{\Gamma}) \quad \mathit{\Gamma},\ @_{\gamma}\ \Diamond\gamma_1,\ \mathsf{x}\colon @_{\gamma_1}\ \mathsf{A} \vdash \mathsf{e}\colon @_{\gamma_1}\ \mathsf{B}}{\mathit{\Gamma} \vdash \underline{\lambda}\mathsf{x.e}\colon @_{\gamma}\ (\mathsf{A}\to\mathsf{B})}\ \text{CAbs}$$

Fig. 6. The rules of type system, interpreted as logic

First, we re-write the typing rules according to the interpretation of types we have just given (writing the types as A, for easier comparison with $\mathcal{NJ}^{\gamma}$'s proof rules): Fig.6. We have inserted the types of the main constants.

The rule (Var) is just (Assm) of $\mathcal{NJ}^{\gamma}$, and (App) is an instance of $\to$E, for the case of γ being γ_B. The rule (CAbs) is $\to$I, whose side-conditions are satisfied given that code types in <NJ> must be simple types and hence in the corresponding $@_{\gamma}$ A, the formula A contains no nominals whatsoever. For (Sub), we have the following derivation in terms of $\mathcal{NJ}^{\gamma}$, keeping in mind that all types of <NJ> correspond to simple formulas:

$$\frac{\dfrac{\mathit{\Gamma} \vdash \mathsf{e}\colon @_{\gamma_1}\ \mathsf{A}}{\mathit{\Gamma} \vdash \mathsf{e}\colon @_{\gamma_1}\ \Box\mathsf{A}}\ \Box\text{I1} \qquad \mathit{\Gamma} \vdash @_{\gamma_1}\ \Diamond\gamma_2}{\mathit{\Gamma} \vdash \mathsf{e}\colon @_{\gamma_2}\ \mathsf{A}}\ \Box\text{E}$$

For (Abs),

$$\frac{\dfrac{\dfrac{\mathit{\Gamma},\ \mathsf{x}{:}@_{\gamma_B}\ \mathsf{A} \vdash \mathsf{e}\colon @_{\gamma_B}\ \mathsf{B}}{\mathit{\Gamma},\ @_{\gamma_B}\ \Diamond\gamma',\ \mathsf{x}{:}@_{\gamma_B}\ \mathsf{A} \vdash \mathsf{e}\colon @_{\gamma_B}\ \mathsf{B}}\ \text{weaken} \qquad \dfrac{\dfrac{}{\mathit{\Gamma},\ @_{\gamma_B}\ \Diamond\gamma',\ \mathsf{x}{:}@_{\gamma_B}\ \mathsf{A} \vdash @_{\gamma_B}\ \Diamond\gamma'}\ \text{Assm}}{\mathit{\Gamma},\ @_{\gamma_B}\ \Diamond\gamma',\ \mathsf{x}{:}@_{\gamma_B}\ \mathsf{A} \vdash @_{\gamma_B}\ \gamma'}\ \text{Nom}}{\mathit{\Gamma},\ @_{\gamma_B}\ \Diamond\gamma',\ \mathsf{x}{:}@_{\gamma'}\ \mathsf{A} \vdash \mathsf{e}\colon @_{\gamma'}\ \mathsf{B}}\ \text{EqL, EqR}}{\mathit{\Gamma} \vdash \lambda\mathsf{x.e}\colon @_{\gamma_B}\ (\mathsf{A}\to\mathsf{B})}\ \to\text{I}$$

The admissibility for (ConstPair) is similar. We assumed Γ_1 to be Γ, $@_{\gamma_B}\ \Diamond\gamma_1$, $@_{\gamma_1}\ @_\gamma\ \mathsf{A}$, $@_{\gamma_1}\ \Diamond\gamma_2$, $@_{\gamma_2}\ @_\gamma\ \mathsf{B}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash @_{\gamma_1}\ @_\gamma\ \mathsf{A}}\ \text{Assm}}{\Gamma_1 \vdash @_\gamma\ \mathsf{A}}\ \text{@E1} \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash @_{\gamma_2}\ @_\gamma\ \mathsf{B}}\ \text{Assm}}{\Gamma_1 \vdash @_\gamma\ \mathsf{B}}\ \text{@E1}}{\Gamma_1 \vdash @_\gamma\ (\mathsf{A} \wedge \mathsf{B})}\ \wedge\text{I}}{\Gamma_1 \vdash @_{\gamma_2}\ @_\gamma\ (\mathsf{A} \wedge \mathsf{B})}\ \text{@I}}{\Gamma,\ @_{\gamma_B}\ \Diamond\gamma_1,\ @_{\gamma_1}\ @_\gamma\ \mathsf{A} \vdash @_{\gamma_1}\ (@_\gamma\ \mathsf{B} \to @_\gamma\ \mathsf{A}{\wedge}\mathsf{B})}\ {\to}\text{I}}{\Gamma \vdash \underline{\mathsf{pair}}\colon @_{\gamma_B}\ (@_\gamma\ \mathsf{A} \to @_\gamma\ \mathsf{B} \to @_\gamma\ \mathsf{A}{\wedge}\mathsf{B})}\ {\to}\text{I}$$

The admissibility of (CApp) is analogous.

Since all type checking rules are admissible, it follows that <$\underline{\texttt{NJ}}$> terms represent proof terms of the $\mathcal{NJ}^\gamma$ formulas expressed by their types.

5 Related Work

Hybrid logic has developed into a vast area, with many different variations, models, axiomatizations, and proof systems. The rather comprehensive overview can be found in [1, 2]. Most of hybrid logic systems are based on classical propositional logic. The rare intuitionistic exceptions are described in [3]. However, the latter system essentially takes the direct product of the hybrid modal semantics (whose worlds interpret, say, time moments) and the Kripke semantics for intuitionistic logic, whose worlds are 'knowledge states'. Correspondingly, Braüner and de Paiva introduce two accessibility relations, for interpretating modal operators and nominals, and for relating knowledge states. Our approach is different: we hybridize standard intuitionistic logic, introducing nominals and satisfaction statement. Our worlds are just knowledge-states, interpreted as future-stage binding environments. (The price we pay is the non-orthodox rules, which Braüner and de Paiva avoid.)

Connections of staging calculi with (non-hybrid) modal logic has been investigated in [4, 5, 13]. They all consider multistage languages and the accessibility relation relating stages. In contrast, <$\underline{\texttt{NJ}}$> is specifically two stage and accessibility relation is between future-stage binding environments.

Following the Lisp tradition, many stage calculi have a special form to 'quote' the code being generated (variously called $\Box$ in $\lambda^\Box$, next in λ°, bracket in MetaOCaml, etc.). Albeit quite convenient to program with, it brings the problem of dealing with free variables within these quotes. As Davies and Pfenning thoroughly discuss in [5], substituting inside quotes is a rather delicate matter. Naive approaches quickly break subject reduction. (Actually, the problem of substituting in modal context was discussed already by Quine, who declared such substitutions nonsensical.) The code combinator approach used in <NJ> (and <$\underline{\texttt{NJ}}$>) avoids this problem entirely, by not having quotes to start with.

Davies and Pfenning [4, 5] stress the importance of so-called 'local soundness' and 'local completeness' when designing the inference rules. Naively, local soundness does not hold for our $\to$I and $\to$E rules. It also does not hold for Braüner and de Paiva's (BoxI) and (BoxE) rules, and for many such rules that introduce eigenvariables.

6 Conclusions

We have pared down the `<NJ>` calculus of code generation with mutable cells to its bare essentials, distilling from it the calculus `<NJ>`. We have proposed a variant of hybrid intuitionistic modal logic $\mathcal{NJ}^\gamma$, with the Kripke knowledge-state–like semantics and proof system. Our $\mathcal{NJ}^\gamma$ is essentially intuitionistic logic with nominals and satisfaction statements borrowed from tense hybrid logic. We then related `<NJ>` and $\mathcal{NJ}^\gamma$, by interpreting `<NJ>` types as logic formulas and demonstrating the admissibility of the calculus typing rules in $\mathcal{NJ}^\gamma$. We have thus advanced the logical understanding of `<NJ>`.

For future work, it is interesting to investigate the generalization of `<NJ>` to multiple stages (at the very least, lifting the restriction that in $\langle \mathsf{s} \rangle^\gamma$, s must be a simple type.)

Bibliography

[1] Carlos Areces and Balder ten Cate. Hybrid logics. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, pages 821–868. Elsevier Science Inc., New York, 2006.

[2] Torben Braüner. Hybrid logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2017 edition, 2017.

[3] Torben Braüner and Valeria de Paiva. Intuitionistic hybrid logic. *Journal of Applied Logic*, 4:231–255, 2006.

[4] Rowan Davies. A temporal logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.

[5] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[6] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Combinators for impure yet hygienic code generation. *Science of Computer Programming*, 112 (part 2):120–144, November 2015.

[7] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers - type- and scope-safe code generation with mutable cells. In Atsushi Igarashi, editor, *APLAS*, volume 10017 of *LNCS*, pages 271–291. Springer-Verlag, 2016.

[8] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3):23:1–49, June 2008.

[9] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. RandIR: differential testing for embedded compilers. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016*, pages 21–30. ACM, October 30 - November 4 2016.

[10] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *ICFP*, pages 217–228, New York, 2010. ACM Press.

[11] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.

[12] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL*, pages 26–37, 2003.

[13] Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. *Logical Methods in Computer Science*, 6(4), 2010.