Partial Type Inference with Higher-Order Types

Paolo Herms

17 July 2009

Abstract

The language MLF is an extension of System F that permits robust first-order partial type inference with second-order polymorphism.

We propose an extension of MLF's graphical presentation with higher-order types and reduction at the type level as in System F_ω. As inference of higher-order types won't be possible in general, the extension introduces System F-style explicit type abstraction and type application.

Abstract

The goal of this work is the basis of a functional programming language whose type system supports partial type inference in presence of higher-order polymorphism. Higher-order types enable to express type operators as functions at the type level and to type expressions that abstract over type operators.

The starting point of this work is MLF, an extension of System F that permits partial type inference with second-order polymorphism. Functions with polymorphic arguments need second-order types. They can be used to encode existential types, polymorphic iterators and monads.

The original, syntactical presentation of MLF is quite complex as it manages a huge amount of technical details and an alternative, graphical presentation has been proposed. Here, types are directed acyclic term graphs with additional binding edges. Type inference is done by solving graphic constraints. This work builds on the graphical presentation extending the constraint system and the solving mechanism.

Like in ML and unlike in System F, polymorphism is inferred in MLF without the need for System F-style explicit type abstractions and type applications. Hence, the language doesn't provide for such constructs. However, inference of higher-order types is undecidable in general and in certain situations it will even be desired to explicitly determine where type operators are abstract and where, on the contrary, the implementation is given, for instance to encode modules. Therefore, a part of the work is to extend MLF with constructs for explicit type abstraction and application. As an intermediate result, we can now type programs where ML-style implicit polymorphism is mixed with System F-style explicit polymorphism.

The other part of the work is the effective inclusion of β-reduction rules into the type system to support functions at the type level. The rules are given in terms of graphical rewrite rules, which are inspired by the λ-calculus implementation in interaction nets. They are added to the MLF equivalence relation on types. Design decisions has been taken to keep the unification algorithm efficient.

The result is MLF_ω, an extension of MLF with support for explicit type abstraction and higher-order types.

Contents

1 Research Context

One of the main reasons of the success of ML is type inference. The programmer does not need to write any type annotation and still his programs are guaranteed type safe. At the same time, ML provides parametric polymorphism. For instance, one can write $\mathsf{let}\, f = \lambda\,(x)\, x\, \mathsf{in}\ (f\,5,\ f\,\text{"5"})$. Here, the function f has the polymorphic type $\forall\alpha.\alpha \to \alpha$, which is once instantiated as $\mathsf{int} \to \mathsf{int}$ and once as $\mathsf{string} \to \mathsf{string}$.

An important restriction of ML-polymorphism is that polymorphic arguments are not supported — ML-polymorphism is limited to rank-1. For instance, the semantically equivalent expression $(\lambda\,(f)\,(f\,5, f\,\text{"5"}))\,(\lambda\,(x)\,x)$ is not typable in ML, as the argument would need to have a polymorphic type. The basic formalism for higher-rank polymorphism is System F, for which type-inference is unfortunately undecidable. That means that the type of every function parameter has to be given manually by type annotations. Type-variables appearing in annotations have to be explicitly introduced by type abstraction and application. For instance, the above example is written

$$(\lambda\,(f : \forall\,(\alpha)\,\alpha \to \alpha)\,(f\ [\mathsf{int}]\ 5, f\ [\mathsf{string}]\ \text{"5"}))\,(\Lambda\,(\alpha)\,\lambda\,(x : \alpha)\,x)$$

Notice the explicit type abstraction $\Lambda\,(\alpha)$ and type applications $[\mathsf{int}]$ and $[\mathsf{string}]$. This way, the type system only has to perform type-checking, which is decidable in System F. However, the need for full type annotations makes System F rather unusable as a programming language.

Finally, a generalization of System F, F_ω allows for first-class type operators, called *higher-order types*, where β-reduction is performed at the level of types. Type operators are first-class if they can be expressed and handled as first-class functions at the level of types. For instance, we can somewhere define $arrow$ as $\lambda\,(\alpha)\,\lambda\,(\beta)\,\alpha \to \beta$ and then write the expression $(\lambda\,(f : arrow\,\alpha\,\beta)\,\lambda\,(x)\,f\,x)\,(\lambda\,(z)\,z)$.[1] Or, using type abstraction, we can write expressions in which type operators are abstracted over, as $list$ in

$$\Lambda\,(list)\,\lambda\,(Nil : \forall\,(\alpha)\,list\,\alpha)\,\lambda\,(Cs : \forall\,(\alpha)\,\alpha \to list\,\alpha \to list\,\alpha)\,Cs\,5\,(Cs\,4\,Nil)$$

System F_ω is the underlying calculus for the current implementation of the module system in OCaml. This module system is implemented at a distinct layer of the rest of the language and it is worth striving for a formalism which integrates type inference with first-class higher-order types.

A lot of research has been done trying to combine higher-rank polymorphism with type inference. As complete type inference has been proven undecidable, such a type inference will necessarily be partial. This means that in any case, some type annotations will be needed to allow inference of the rest of the program. The several research results differ in how much annotation is needed in programs and where.

[1] Here, the free type variables mean *any type* which corresponds to the semantic of a free type variable in an OCaml phrase. The correct notation for this is $\exists\alpha.\alpha$ but a consequent adoption of this notation would be too heavy-handed.

A first approach is called *Local Type Inference*. Here, local means that missing annotations are recovered using only information from adjacent nodes in the syntax tree. It is an algorithmically driven approach that lacks a clear annotation rule, that is a rule that tells the programmer where type annotations have to be given. Furthermore, it is unstable to program transformations. The current support for rank-2 polymorphism in Haskell is based on this approach.

Currently, the most exhaustive solution to this problem is MLF.

1.1 MLF

The very problem for type inference in System F is the lack of *principal types*. In ML, each sub-expression of a program has a type, that is more general than every other type the expression may be assigned to. This does not work in general in System F, as illustrated by the following example. Consider the function choose of type $\forall(\beta)\,\beta \to \beta \to \beta$ that takes two arguments and returns either one. If choose is applied to the identity function id, with the type $\forall(\alpha)\,\alpha \to \alpha$, this should have a type like $t \to t$ where both occurrences of t are the same instance of $\forall(\alpha)\,\alpha \to \alpha$. Now, we have two options. We can introduce a new type variable γ and instantiate both occurrences of t as $(\gamma \to \gamma)$ resulting in the type $\forall(\gamma)\,(\gamma \to \gamma) \to (\gamma \to \gamma)$. Or, we can leave both of them polymorphic and obtain the type $(\forall(\alpha)\,\alpha \to \alpha) \to (\forall(\alpha)\,\alpha \to \alpha)$. Now, neither of these types is more general than the other one. The inner polymorphism of the second one cannot be recovered by instantiating the first one. Conversely, the first one cannot soundly be instantiated any further. The crucial information that the two instances of $(\forall(\alpha)\,\alpha \to \alpha)$ are linked, and that instantiating them together would be sound, has been lost and we find ourselves with two non confluent strategies: instantiate as soon as possible or instantiate as late as possible. This is fatal for type inference, where once taken decisions are difficult to backtrack.

MLF resolves this problem by enriching the type system with two new forms of bounded quantification. For instance, choose id can receive the type $\forall(\gamma \geq \forall(\alpha)\,\alpha \to \alpha)\,\gamma \to \gamma$, read $\gamma \to \gamma$, where γ is an instance of $\forall(\alpha)\,\alpha \to \alpha$. This way we can always decide to instantiate as late as possible, keeping the several instances synchronized. This form of quantification is called *flexible*, because the bound can freely be instantiated further. For instance, if choose id is applied to succ with the type $\mathsf{int} \to \mathsf{int}$, then the $\forall(\alpha)\,\alpha \to \alpha$ in $\forall(\gamma \geq \forall(\alpha)\,\alpha \to \alpha)\,\gamma \to \gamma$ becomes $\mathsf{int} \to \mathsf{int}$ resulting into

$$\mathsf{choose\ id\ succ} : \forall(\gamma \geq \mathsf{int} \to \mathsf{int})\,\gamma \equiv \mathsf{int} \to \mathsf{int}$$

The counterpart of flexible quantification is *rigid* quantification. It is introduced in MLF to be able to perform inference in the presence of type annotations. For instance, the expression $\lambda(z : \forall(\alpha)\,\alpha \to \alpha)\,z\,z$ receives the type $\forall(\beta' = \forall(\alpha)\,\alpha \to \alpha)\,\forall(\beta'' \geq \forall(\alpha)\,\alpha \to \alpha)\,\beta' \to \beta''$ with the precise meaning that a polymorphic function is required as parameter.

Even if these types may seem rather unappealing, the result is comforting. MLF has a clear and simple annotation rule:

Only function parameters have to be annotated, in particular if and only if they are used in a polymorphic way.

For instance, the auto-application $\lambda\,(z)\,z\,z$ we just saw, must be annotated as z is used polymorphically, while the identity function need not. So you can write $(\lambda\,(z : \forall\,(\alpha)\,\alpha \to \alpha)\,z\,z)\,(\lambda\,(x)\,x)$ which is typed correctly as the identity automatically receives the polymorphic type $\forall\,(\alpha)\,\alpha \to \alpha$.

Therefore, MLF is a conservative extension of ML. In particular, all ML terms are typable in MLF. At the same time, it offers the full power of System F without the need for explicit type abstraction and type application.

Moreover, the set of well-typed programs is invariant under a wide class of program transformations, including let-expansion, let-reduction, η-expansion, reordering of arguments, curryfication, and also "abstraction of applications", which means that $a_1\,a_2$ is typable if and only if $\mathsf{apply}\,a_1\,a_2$ is, with apply defined as $\lambda\,(x)\,\lambda\,(x)\,f\,x$.

Recall that our goal is to find a way to combine (partial) type inference with higher-order types. Apart from the *kinding* problem, that is, the well-formedness of higher-order type expressions, the F_ω type equivalence rules which implement β-reduction could, in principle, be included unchanged into a syntactical presentation of MLF with a reintroduced form of explicit quantification. However, the original syntactical presentation of MLF is too technical to enable checking the formal details of an extension with a reasonable effort.

1.2 HML

One possible approach is to start the work on the simpler and less ambitious version HML, which has a relatively simple syntactical presentation. HML avoids the use of rigid quantifiers by just inlining rigidly bound types, which leads to a loss of information about sharing and as a result requires more type annotations compared to MLF. The stronger, yet simple, annotation rule in HML is: Function parameters *with a polymorphic type* must be annotated. Recall that in MLF function parameters must be annotated only if they are *polymorphically used*.

For instance, recall that both type systems reject the self application ω, defined as $\lambda x.x\,x$, because x is used polymorphically, unless it is annotated as in the expression $\lambda\,(x : \forall\alpha.\alpha \to \alpha)\,x\,x$ (referred to as $\omega^\dagger$) which is accepted by both type systems. Now, the expression $\lambda x.\omega^\dagger\,x$ is accepted only by MLF while HML would reject it, as the parameter has a polymorphic type, though not used polymorphically. Notice that $\lambda x.\omega^\dagger\,x$ is the η-expansion of $\omega^\dagger$ and the β-reduction of $\mathsf{apply}\,\omega^\dagger$, which are both accepted by HML, where apply stands for the expression $\lambda x.\lambda y.x\,y$.

This shows how the seemingly little difference in the annotation rule leads HML to be less robust to small program transformations. We preferred therefore to build on the graphical presentation of MLF, which is more general and well-understood, to extend it with higher-order types.

2 Graphic types and constraints

2.1 ML Graphic types

ML graphic types are directed acyclic term graphs. Every node is labeled with a symbol. Variable nodes are labeled with the pseudo symbol $\perp$, all the other symbols stand for type operators they represent. At least the arrow $\rightarrow$ has to figure among type constructors. Graphic types are generally graphs and not trees because sub-graphs can be shared. Sharing of variable nodes is important to mean several occurrences of a type variable within a type, while sharing of inner nodes as in τ_3' in Figure 1 is not semantically significant in ML.[2] That's the reason why ML graphic types can be printed out as trees and read back as graphs. For instance, τ_2, which represents the ML type $(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$, is different from τ_3 because the $\perp$-node is shared in the latter. On the contrary, τ_3 and τ_3' are equivalent to represent both the ML type $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$.

In ML, the type $(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$ is an instance of $\alpha' \rightarrow (\beta \rightarrow \beta)$, graphically represented by τ_1, that is, it is less general because the type variable α' has been substituted or *instantiated* by the type $\alpha \rightarrow \alpha$, which in turn could be further instantiated to, say $\mathsf{int} \rightarrow \mathsf{int}$. This is captured by the well-defined instance relation on ML types. The instance relation on ML graphic types reflects the instance relation on ML types, but is a little more fine-grained. In particular $\tau_1 \leq \tau_2 \leq \tau_3 \leq \tau_3'$ holds but $\tau_3' \leq \tau_3$ does not. Instance on ML graphic types is defined as the union the two atomic operations *grafting*, i.e. the substitution of a $\perp$-node by a sub-graph, and *merging*, the fusion of two isomorphic sub-graphs. In Figure 1, τ_3' is an instance of τ_3 by merging, which is an instance of τ_2 by merging, which in turn is an instance of τ_1 by grafting. Notice how, syntactically, grafting corresponds to substituting the occurrences of a type variable with a type and merging two $\perp$-nodes corresponds to substitute the occurrences of a type variable by another type variable. For example, $(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$ becomes $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ when merging α and β. Merging two inner nodes works similarly, except that it has no syntactical correspondence, thus the strict inclusion of the instance relations. The fact that instantiating τ_3 to τ_3' is somehow *reversible* is captured by the reflexive similarity relation $\approx$ on graphic types, i.e. $\tau_3 \approx \tau_3'$.

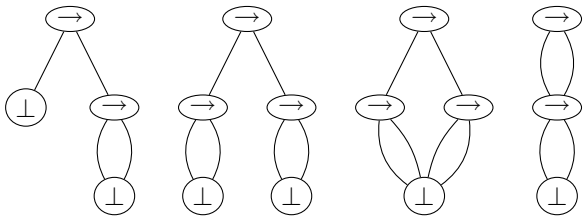


Figure 1: ML Graphic Types

[2]Implementations of type inference may introduce sharing of inner nodes for efficiency

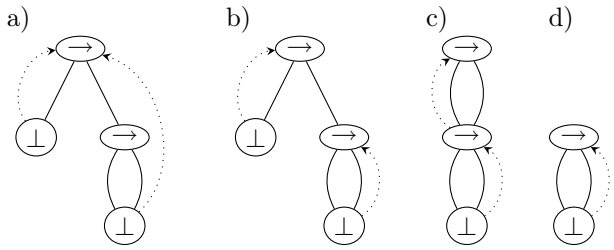


Figure 2: Polymorphic Graphic Types

2.2 Polymorphic Graphic Types

In ML, polymorphism is prenex or rank-1. That is, what an ML type like $\alpha \to (\beta \to \beta)$ really means is $\forall\,(\alpha)\,\forall\,(\beta)\,\alpha \to (\beta \to \beta)$. Binders can be omitted in ML, because they would always appear at top-level. In higher-rank type systems, polymorphic types can appear under type constructors as in $\forall\,(\alpha)\,\alpha \to (\forall\,(\beta)\,\beta \to \beta)$. Graphically, this is expressed by a binding edge, from the variable node to the node in front of which the quantifier is introduced. See for instance the graphic representations of the last two types, we talked about, in Figure 2 a) and b).

While the first two graphic types have a direct correspondence in syntactic types, this is not the case for the third one. Recall that sharing of inner nodes is not significant in ML, so this could be similar in System F. On the contrary, the essence of MLF is precisely the possibility to represent sharing of synchronized instances, like in the choose id example. Indeed, Figure 2 c) shows the representation of the type of the expression choose id: $\forall\,(\gamma \geq \forall\,(\alpha)\,\alpha \to \alpha)\,\gamma \to \gamma$. Notice the binding edge from the inner arrow-node to the topmost one. It represents the fact that γ is introduced at the outermost position. At the same time, γ is constrained to be an instance of $\forall\,(\alpha)\,\alpha \to \alpha$. This is exactly represented by the sub-graph under the first $\to$. For comparison, see Figure 2 d) which shows the representation of $\forall\,(\alpha)\,\alpha \to \alpha$. Indeed, if no constraint is given in a quantification as in $\forall\,(\alpha)\,\forall\,(\beta)\,\alpha \to (\beta \to \beta)$ it actually means that the variable is an instance of $\bot$, which is always true. In full, this type could be written $\forall\,(\alpha \geq \bot)\,\forall\,(\beta \geq \bot)\,\alpha \to (\beta \to \beta)$ and that is also the reason why unconstrained variable nodes are labeled with $\bot$.

While two instance operations are sufficient to describe the instance relation on ML graphic types, another two are needed for MLF to operate on the binding tree: *Raising* of a binder extrudes polymorphism as in $\forall\,(\alpha)\,\alpha \to (\forall\,(\beta)\,\beta \to \beta)$ where raising the variable β results in $\forall\,(\alpha)\,\forall\,(\beta)\,\alpha \to (\beta \to \beta)$. *Weakening* turns a flexible binding edge into a rigid one. Figure 3 shows the four atomic instance operations in detail. The instance relation for MLF is then defined as the transitive union of the four atomic operations. The fact that some operations may not be allowed for certain nodes in a graphic type is expressed by a permission system. Roughly, rigidly bound nodes (permission R) and their sub-graphs (permission L) are protected from instantiations such that required

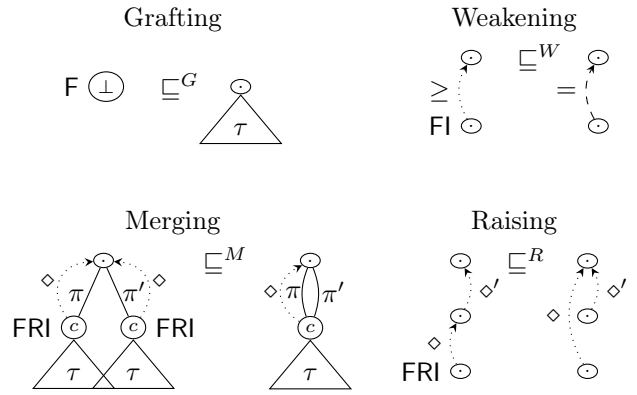


Figure 3: Instance operations

polymorphism is preserved, whereas flexibly bound nodes (permission F) can be freely instantiated. The instance relation for MLF, written$\sqsubseteq$, is then defined as the transitive union of the four atomic operations $\left(\sqsubseteq^G \cup \sqsubseteq^W \cup \sqsubseteq^M \cup \sqsubseteq^R\right)^*$.

2.3 Graphic Constraints

Graphical type inference is based on graphic constraints. To perform type inference of an expression, the algorithm generates a graphic constraint by structural translation of the expression's syntax tree and then solve it. For the algorithm, a graphic constraint is solved, when is doesn't contain any more constraint edges. Therefore, the algorithmic solution of a graphic constraint is a graphic type - a type which can be assigned to the expression.

We consider two types of constraint edges: unification edges and instantiation edges.

- A unification edge n_1 ⟞ n_2 links two type nodes and means that the types under n_1 and n_2 should become equal. It is solved and can be removed when n_1 and n_2 can be merged according to the previously defined instance relation. Of course, this may require other instance operations before a merge is possible.

- An instantiation edge g ⇢ n relates a G-node g to a type node n. It requires the type under n to be an instance of the type scheme represented by g. It is solved by adding to the constraint an expansion of the type scheme and a unification edge between its root node and n. This way, the problem of resolving an instantiation edge is reduced to the resolution of a unification edge. The expansion of a type scheme is, roughly, a copy of the type under the G-node.

Each sub-expression is typed at its generalization level at which polymorphism can be introduced. Graphically, an expression is translated into a constraint

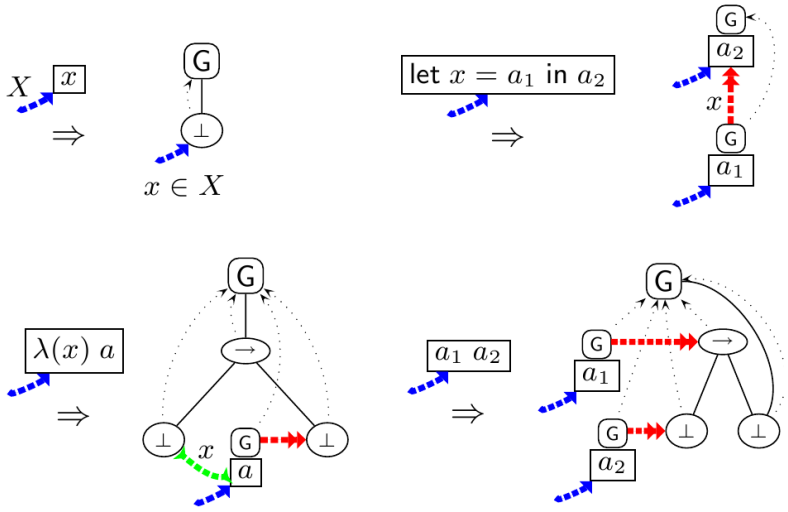


Figure 4: Translation of expressions into graphic constraints

graph with a G-node as root, at which every node in the constraint is initially bound. If the expression is a composition of sub-expressions, these sub-expressions are recursively translated and their translations' roots, which in their turn are again G-nodes, are all bound to the main expressions root node, too. See Figure 4 for the precise translation rules. The rectangular boxes stand for the result of the translation function applied to the expression they are labeled by. Expressions are translated under a typing environment that contains constraint edges, each of which labeled by the variable it constraints if it appears free within the expression.

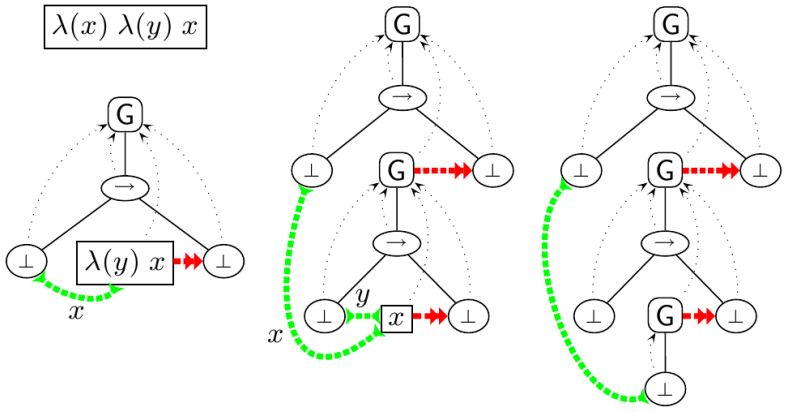


Figure 5: Translation of $\lambda\,(x)\,\lambda\,(y)\,x$

As an example, we show the type inference of the expression $\lambda\,(x)\,\lambda\,(y)\,x$. See at first its incremental translation into a graphic constraint in Figure 5. After two applications of the abstraction rule and one application of the variable rule, the simple expression is completely translated. Notice how the unification edge has the meaning that the codomain of the first arrow should have the same type as the occurrence of x in the body. The codomain of the second arrow hasn't any outgoing unification edge because y doesn't appear in the body.

Figure 6 shows the evolving inference algorithm on $\lambda\,(x)\,\lambda\,(y)\,x$. The first picture shows the actually generated type constraint out of the expression. Here, an optimization has been applied, which considers that a type scheme with only a single $\bot$-node can directly "pass through" an incoming unification edge. In the second picture, the unification edge has been solved as the two linked nodes are merged. In the third picture, the type scheme that constrains the domain of the first $\rightarrow$-type has been *expanded*, that is a sort of *working copy* to unify with has been added. Notice how the node outside the sub-graph's scope is shared between the copy and the original graph. Having performed this expansion, the instantiation edge is solved and can be removed - so can all the nodes which haven't got any incoming constraint edges and are not in the root node's structural sub-tree, that is have become orphans. The fourth picture shows the cleaned up constraint, which is further instantiated in the fifth picture, in order to merge the two linked nodes in the last picture. The constraint has now been solved and therefore corresponds to the principal type $\forall\,(\alpha)\,\alpha \rightarrow \forall\,(\beta)\,\beta \rightarrow \alpha$ of the expression $\lambda\,(x)\,\lambda\,(y)\,x$.

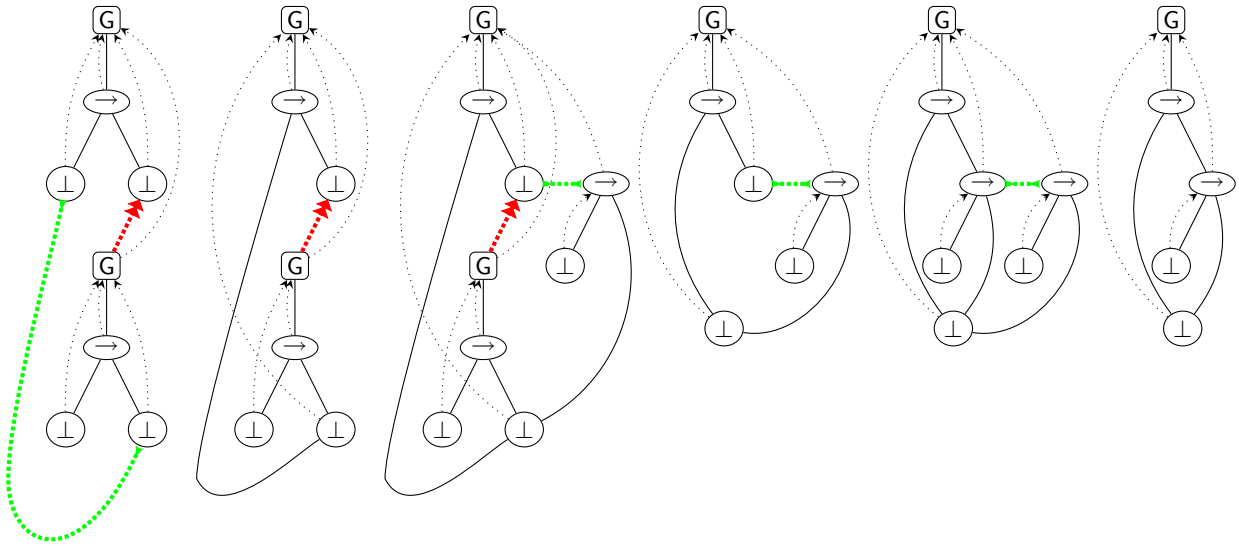


Figure 6: Type Inference of $\lambda\,(x)\,\lambda\,(y)\,x$

3 Higher-Order Graphic Types

The goal of this work is to add support for higher-order types to the graphical presentation of MLF to obtain a language with first-class type-operators in which most of the types can be inferred. MLF infers polymorphism and implicitly instantiates polymorphic types when needed and that way avoids completely explicit type abstractions and type applications. However, the sense of first-class type-operators is precisely to be able to write expressions that abstract over the implementation of a type operator in order to apply them later to its implementation. OCaml modules are modeled that way. Therefore, our language needs a form of System-F-like explicit type abstraction and application.

3.1 Explicitly-Bound Types

Explicit type abstraction cannot be handled with the original MLF binding mechanism. If an explicit type variable was represented graphically as just another flexibly bound bottom node, it could be implicitly instantiated during type inference and the following type annotation would hold (the expression would be typable): $(\Lambda(\alpha)\,\lambda(x:\alpha)\,x) : \mathsf{int} \to \mathsf{int}$ (see Figure 7). This would be counter intuitive as in System F the type of the left hand side of the annotation is $\forall\alpha.\alpha \to \alpha$ which is polymorphic, though not equal to $\mathsf{int} \to \mathsf{int}$ and can become $\mathsf{int} \to \mathsf{int}$ only by explicit type application $(\Lambda(\alpha)\,\lambda(x:\alpha)\,x)\,[\mathsf{int}]$.

Another difference with implicit polymorphism concerns the order of introduction of type variables. The two types $\forall(\alpha)\,\forall(\beta)\,\alpha \to \beta \to \beta$ and $\forall(\beta)\,\forall(\alpha)\,\alpha \to \beta \to \beta$ are equivalent in MLF and have the same graphic representation. This must not be the case if these type variables are introduced explicitly as in $\Lambda(\alpha)\,\Lambda(\beta)\,\lambda(x:\alpha)\,\lambda(y:\beta)\,y$ and $\Lambda(\beta)\,\Lambda(\alpha)\,\lambda(x:\alpha)\,\lambda(y:\beta)\,y$. Since the order of type application matters, these two terms must have different types.

This gives the intuition that implicit and explicit type abstraction should be reflected differently in types. We'll give them the syntactic types $\forall^i(\alpha)\,\alpha \to \alpha$ and $\forall^e(\alpha)\,\alpha \to \alpha$ so as to distinguish whether the type variable was introduced implicitly or explicitly. ML and MLF types contain only the implicit version, whereas all System F types are of the explicit one. In MLF^ω both will occur so we need the two different syntactic forms simultaneously in the language.[3] With this notation the two types $\forall^i(\alpha)\,\forall^i(\beta)\,\alpha \to \beta \to \beta$ and $\forall^i(\beta)\,\forall^i(\alpha)\,\alpha \to \beta \to \beta$ are equivalent and $\forall^e(\alpha)\,\forall^e(\beta)\,\alpha \to \beta \to \beta$ and $\forall^e(\beta)\,\forall^e(\alpha)\,\alpha \to \beta \to \beta$ are different.

3.2 Explicitly-Bound Graphic Types

3.2.1 Explicit binding edges

The adopted solution consists in introducing a new type of binding edge, the explicit binding edge . (See Figure 8 for an example.) It stands for a

[3] Of course, this is just one possible notation. In a programming language, one could prefer a notation where the implicit binder is notated simply $\forall$ and only the explicit binder, which will occur much less frequently, is marked specially, like $\overline{\forall}$ for instance.

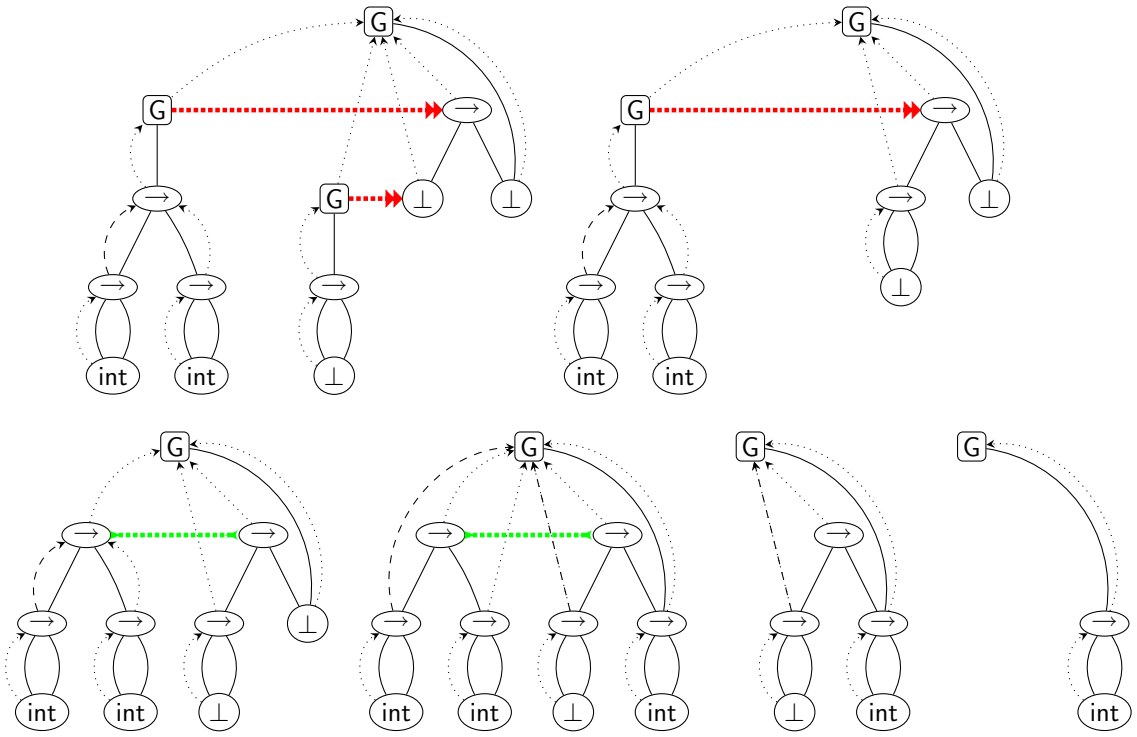


Figure 7: Inference of $(\lambda\,(x : \alpha)\,x) : \mathsf{int} \to \mathsf{int}$

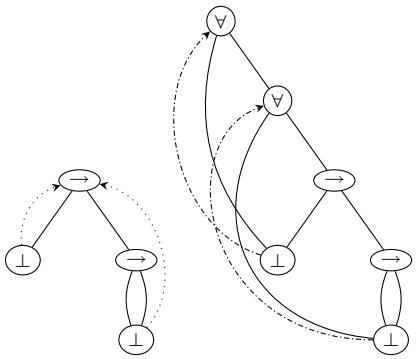


Figure 8: $\forall^i\alpha.\forall^i\beta.\alpha \to \beta \to \beta$ and $\forall^e\alpha.\forall^e\beta.\alpha \to \beta \to \beta$

delayed flexible edge, that is, an edge that can become flexible at a later point but that prevents any instantiation till then. To distinguish the order in which explicit variables are introduced, the corresponding nodes are bound on ∀-nodes. We choose the symbol ∀ to be of arity 2, where such a ∀-node's first child is always the variable node bound on that ∀-node. This way, there will always be a structural edge *and* a binding edge between a ∀-node and its variable node, which will help with type inference.

3.2.2 Constraint edges

The function of the new *explicit* binding edge is to prevent explicitly introduced type variables from being instantiated during type inference. For instance, in Figure 7, if the left-hand bottom node was explicitly bound, the unification edge could not be solved, as this would require to graft the type int into the bottom node. This resolves the issue of *implicit* instantiation of explicit type variables. However, we still have to be able to instantiate them *explicitly*, that is, to type an expression like $a\,[\tau]$ by replacing α in the type of a which must be of the form $\forall^e\alpha.\tau'$.

For this purpose, we define a new *explicit* instantiation edge $g \bullet\!\!\cdots\!\!\twoheadrightarrow n$ that relates a G-node g to a type node n. Like the original instantiation edge, it requires the type under n to be an instance of the type scheme represented by g with, and this is new, the top-most explicit binding edge transformed into an ordinary flexible binding edge.

3.2.3 Translation

Type inference for expression terms is done starting from the structural translation of these terms into typing constraints. If expressions are extended with explicit type abstraction and explicit type application as in System F, we have to add two new rules to the translation function.

$$a ::= \ldots \mid \Lambda\alpha.a \mid a\,[\tau] \quad {}^{4}$$

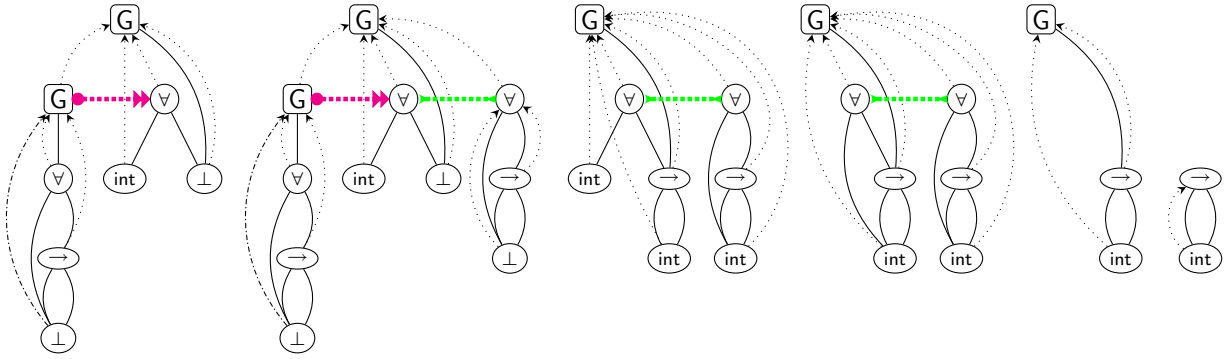


Figure 9: Constraint resolution for $(\Lambda\alpha.\lambda x\!:\!\alpha.x)\,[\mathsf{int}]$

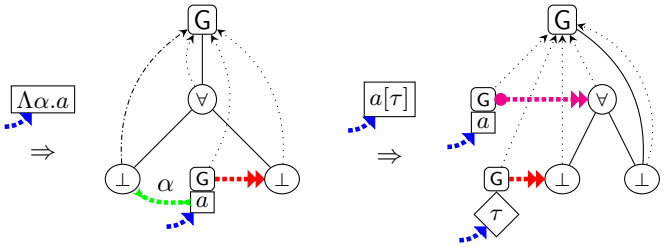


Figure 10: Translation rules for explicit type abstraction and application

The two new rules are shown in Figure 10. Recall that the rectangular boxes stand for the result of the translation function applied to the expression they are labeled by, and that expressions are translated under a typing environment that contains constraint edges, each of which labeled by the variable it constraints if it appears free within the expression.

- A type abstraction $\Lambda\alpha.a$ is typed as a type scheme containing an explicitly polymorphic type $\forall^e\alpha.\tau$, where τ must be an instance of the type of a. The free variables of a are constrained by the typing environment, except for occurrences of the type variable α which must be unified with the explicitly bound type variable of the polymorphic type.

- A type application $a\,[\tau]$ is typed as $\forall^i\,(\alpha \geq \tau')\,\tau''_\alpha$, where τ' must be an instance of τ and where τ''_α, which generally contains free occurrences of α, must be such that $\forall^e\alpha.\tau''_\alpha$ is an instance of the type of a.
  More precisely, in terms of graphic types, the resulting type is the second child of an existentially introduced $\forall$-node, which is constrained to be an *explicit instance* of the type of a, and whose first child is constrained to

[4] where α ranges over type variables and τ ranges over types

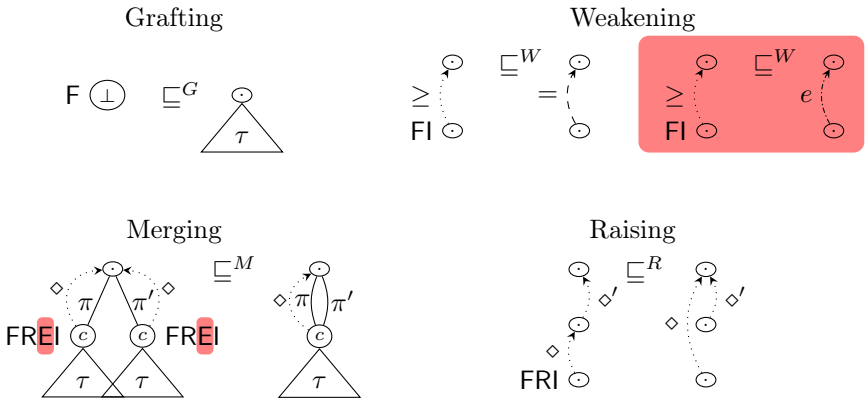


Figure 11: Instance operations, extensions wrt. MLF emphasized

be an instance of τ. The free variables of a and the free type variables of τ are constrained by the typing environment.

3.2.4 The instance relation

The new instance relation on MLF^ω is still defined as the composition of the four atomic instantiation steps Grafting, Weakening, Merging and Raising, that is, $\sqsubseteq$ is defined as $\left(\sqsubseteq^G \cup \sqsubseteq^W \cup \sqsubseteq^M \cup \sqsubseteq^R\right)^*$. See the slightly modified instance operations in Figure 11. The addition of the new binding edge introduces very little change, as we have to forbid most instance operations for explicitly bound nodes, as explained above. To this effect, we assign every explicitly bound node the new permission E, for *explicit*. Only Merging is allowed for such E-nodes.

Recall that in general Merging requires the two sub-graphs being fused to be isomorphic and to be bound at the same position with the same kind of binder. Thus, the only way to unify an E-node with another node is to raise the latter node until it is bound at the same position as the E-node and then to weaken it to explicitly bound, the reason for which the Weakening operation has been extended. As E-nodes cannot be raised, this is only possible if the node to unify with is bound at a lower position, so we shall type neither $\lambda\,(x : \alpha)\,\Lambda\,(\beta)\,\lambda\,(y : \beta)\,x = y$, as this would require unifying α and β, nor $\Lambda\,(\alpha)\,\Lambda\,(\beta)\,\lambda\,(x : \alpha)\,\lambda\,(y : \beta)\,x = y$. Furthermore, as Grafting is not allowed and as the translation function generates typing constraints in which all explicitly bound nodes are $\bot$-nodes, we won't type $(\Lambda\,(\alpha)\,\lambda\,(x : \alpha)\,x = 5)$ either, as this would require unifying int with α, whose permission is E and E does not allow grafting.

All this has the desired effect of not implicitly instantiating explicit type variables. On the other hand, see the example in Figure 12 of how partial inference correctly works under an explicit binder.

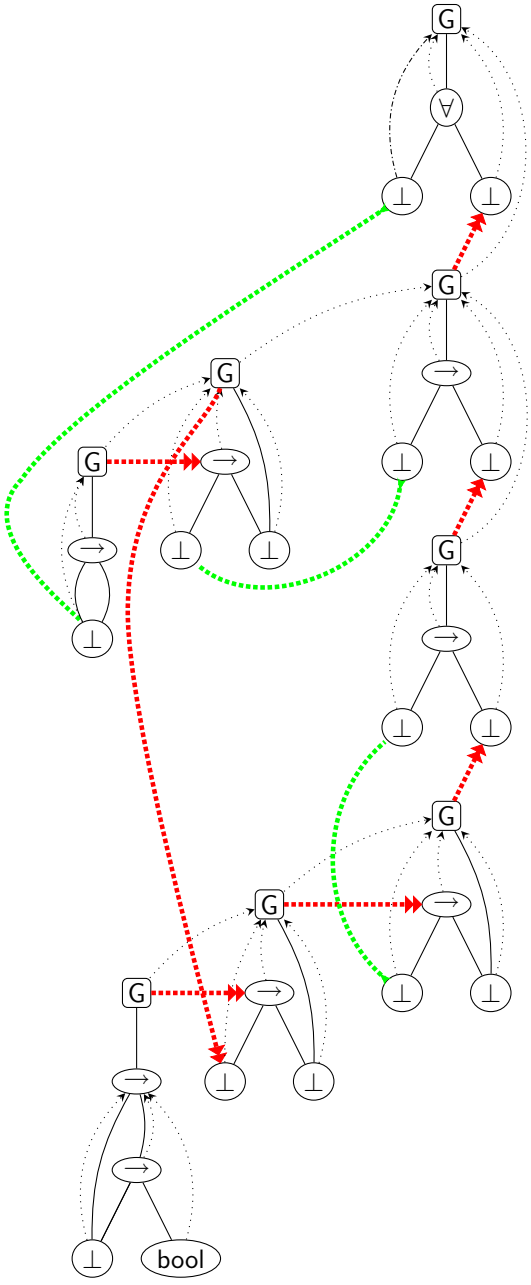


Figure 12: Inference of $\Lambda\,(\alpha)\,\lambda\,(x:\alpha)\,\lambda\,(y)\,x = y$

3.2.5 Conversion

It might be useful to switch an expression from explicitly typed to implicitly typed or vice versa. For the former, we just have to apply a Λ-expression to a fresh[1] type variable; For the latter, we have to introduce a type variable with a Λ and annotate the expression with a monomorphic type using the new type variable. See the following examples for explanation.

$$\text{Assume} \quad \overline{\mathsf{a}} \equiv \Lambda\gamma.\Lambda\delta.\lambda\,(x:\gamma)\,\lambda\,(y:\delta)\,y \quad \text{and} \quad \mathsf{a} \equiv \lambda x.\lambda y.y \quad \text{then}$$

$$\begin{aligned}
\overline{\mathsf{a}} &: \forall^e\gamma.\forall^e\delta.\gamma \to \delta \to \delta \\
\overline{\mathsf{a}}\,[\exists\alpha'.\alpha']\,[\exists\beta'.\beta'] &: \forall^i\alpha.\forall^i\beta.\alpha \to \beta \to \beta \\
\mathsf{a} &: \forall^i\alpha.\forall^i\beta.\alpha \to \beta \to \beta \\
\Lambda\gamma\Lambda\delta.\,(\mathsf{a}:\gamma \to \delta \to \delta) &: \forall^e\gamma.\forall^e\delta.\gamma \to \delta \to \delta
\end{aligned}$$

One could think about using some syntactic sugar for the conversions, like $\overline{\mathsf{a}}\,[]\,[]$ for $\overline{\mathsf{a}}\,[\exists\alpha.\alpha]\,[\exists\beta.\beta]$ or $\mathsf{a} :!\,\alpha,\beta.\alpha \to \beta \to \beta$ for $\Lambda\gamma\Lambda\delta.\,(\mathsf{a}:\gamma \to \delta \to \delta)$. What is remarkable is that we don't need any new construct in the language.

It might also be useful to change the order of explicit type abstractions of an expression. For that purpose, we just have to abstract type variables in the desired order and apply them in the old order to the expression, as in the following example.

$$\begin{aligned}
\overline{\mathsf{a}} &: \forall^e\gamma.\forall^e\delta.\gamma \to \delta \to \delta \\
\Lambda\delta'.\Lambda\gamma'.\overline{\mathsf{a}}\,[\gamma']\,[\delta'] &: \forall^e\delta'.\forall^e\gamma'.\gamma' \to \delta' \to \delta'
\end{aligned}$$

3.3 Higher-order types

To represent higher-order types graphically, we introduce two new symbols, λ for abstraction and @ for application, both of which have arity 2. With these new symbols, type expression terms can, just as before, be translated into directed acyclic term graphs just by sharing occurrences of the same variable in the term tree. Here, λ-introduced variable nodes are bound to the corresponding λ-node. Precisely, these bindings are explicit, as for explicitly introduced type variables in expressions, so that the left child of a λ-node is ensured to always be a $\perp$-node. For instance, the term tree of the type expression $(\lambda\alpha.\lambda\beta.\alpha\,\beta\,\beta)\,(\lambda\alpha.\lambda\beta.\alpha * \beta)$ is shown in Figure 13 a). This type expression corresponds to the application of a type operator to the pair type operator. This, given a generic type-operator with two parameters, returns another type-operator with one parameter, that applies its argument twice to the original type-operator. So, the result of the application in the example is a type operator, which given a type, returns the type of a pair of that type.

Now, somewhere else in a program, we could have written the expression $(5, 4)$ with the obvious type $\mathsf{int} * \mathsf{int}$. Can we give to the expression also the type $(\lambda\alpha.\lambda\beta.\alpha\,\beta\,\beta)\,(\lambda\alpha.\lambda\beta.\alpha * \beta)\,\mathsf{int}$?

In System F_ω the two types are in the equivalence relation $\equiv$. One of the F_ω-typing rules says that if an expression e has a type τ, which is equivalent to a type

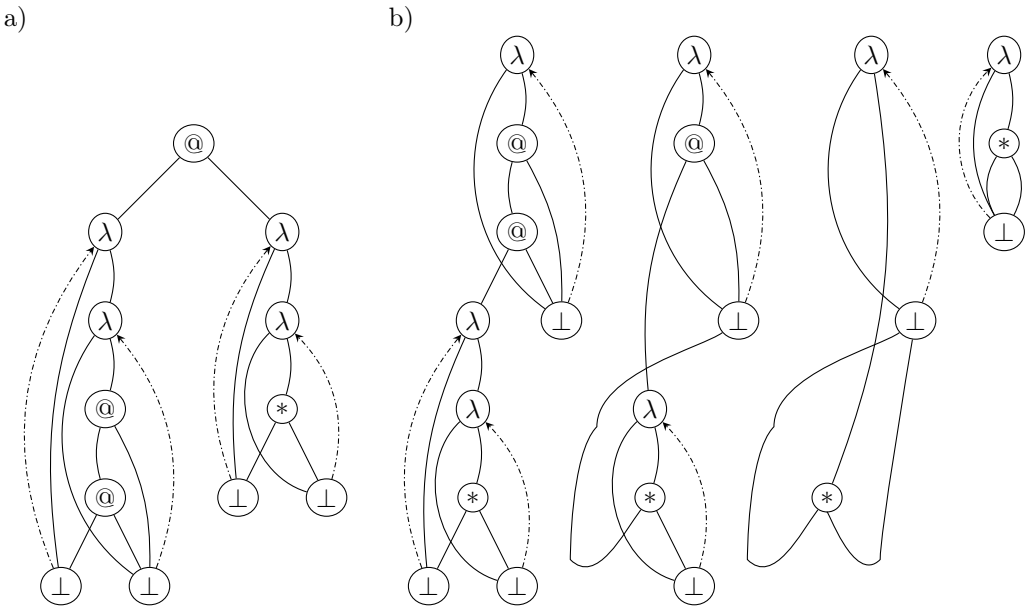


Figure 13: Representation and reduction of $(\lambda\alpha.\lambda\beta.\alpha\,\beta\,\beta)\,(\lambda\alpha.\lambda\beta.\alpha * \beta)$

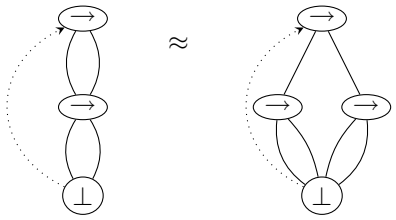


Figure 14: Type equivalence: MLF example

τ', then e also has the type τ'. This relation is defined by type equivalence rules, the most important being the β-reduction rule $(\lambda\alpha.\tau)\,\tau' \equiv \tau\,[^{\alpha}/_{\tau'}]$. Graphically, this rule can be expressed as in Figure 15. The reduction of the example is shown in Figure 13 b). Here, you can see three times the application of the β-rule. Note that in Figure 13 b), the whole graph is shifted upwards comparing to Figure a) and the new root is the node that in Figure a) was the root's left child. The last picture is only a rearrangement of the second last one.

As explained in section 2.1, MLF also has an equivalence relation, $\approx$, which captures sharing and unsharing of *inert* nodes, that is, nodes at which no $\bot$-node is transitively, flexibly bound.[5] For instance, the two types in Figure 14

[5]The fact that equivalence by $\approx$ is restricted to inert nodes wasn't explained in section 2.1, where we didn't consider bindings, yet. However this is a detail that doesn't bother us, as we assure by construction that λ-nodes are always inert.

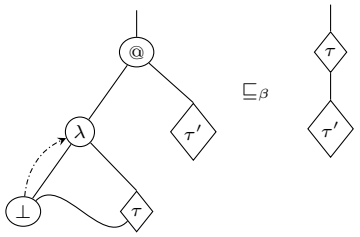


Figure 15: Type equivalence: reduction of a β-redex

are equivalent by $\approx$ in MLF, because the middle node is inert. Precisely, this wouldn't be true if the $\bot$-node was bound to that node, instead of being bound to the root node.

We have to be careful, though, in simply adding a β-reduction rule to the equivalence relation. An important property for the decidability of MLF type inference, is that reversible instance steps always commute to the right with non-reversible ones. That means that it is possible to instantiate a type τ to a type τ' by a sequence of reversible and non-reversible instance steps, if and only if it is also possible to perform that instantiation by first making all the non-reversible steps and only then the reversible ones. Formally $\tau \leq \tau'$ if and only if $\tau \, (\sqsubseteq \cup \approx)^* \, \tau'$ if and only if $\tau \sqsubseteq \tau'' \approx \tau'$ for some τ''. This property allows for an algorithm that performs only non-reversible steps never having to backtrack. In particular, to perform unification of two types τ_1 and τ_2, the algorithm finds a type τ_3 such that $\tau_1 \sqsubseteq \tau_3$ and $\tau_2 \sqsubseteq \tau_3$.

This will no longer work in presence of β-redexes, which may need to be reduced in order to unify two types. Furthermore, consider the type $(\lambda\delta.\delta\,\mathsf{int} * \delta\,\mathsf{bool})\,(\lambda\alpha.\alpha)$ after its first reduction, graphically shown in Figure 16 a). Here, the sub-graph representing $\lambda\alpha.\alpha$ must be duplicated in order to be applied twice. In theory, this is allowed, because a λ-node is always *inert* and can thus be unmerged, but such an operation was not necessary before within the type inference algorithm.

Another difficulty we have to take care of is confluence of reduction strategies, which is necessary for principality of unification. Consider the example in Figure 17. Here, two sub-graphs that contain a redex are unified in a constraint. In particular the redexes discard their arguments. The solution of this constraint depends on whether the unification edge is solved first (Figure 17 a) or the redex is reduced first. If the unification edge is solved first, the two $\bot$-nodes have to be merged (Figure 17 b). This is not necessary if the redexes are reduced first, because the two $\bot$-nodes don't appear linked any longer. Therefore, the solution found in b) is not principal.

The correct strategy is to first reduce the redexes appearing within the sub-graphs to be unified before instantiating further. To verify that this strategy

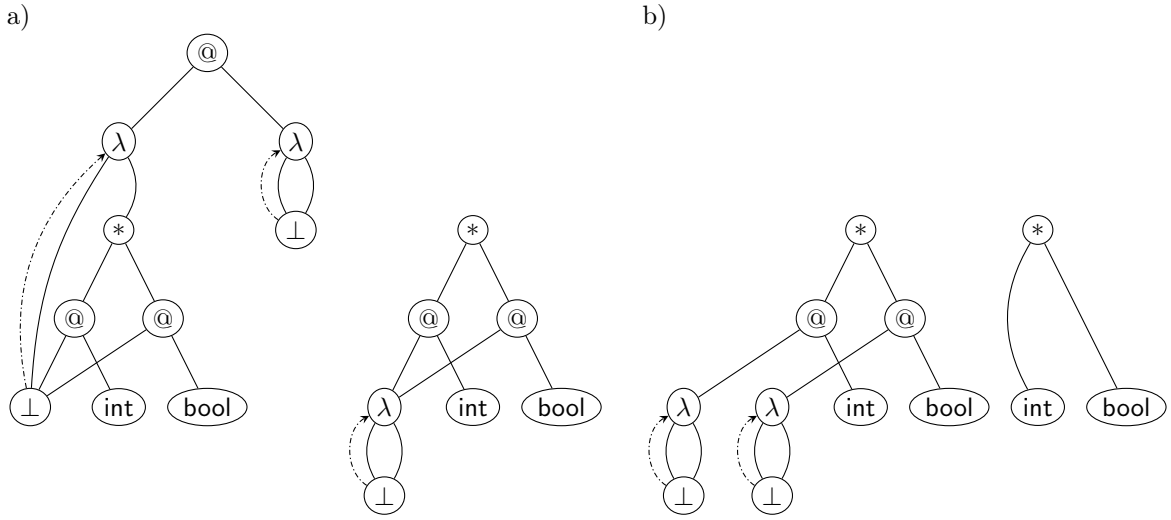


Figure 16: Reduction of $(\lambda\delta.\delta\,\mathsf{int} * \delta\,\mathsf{bool})\,(\lambda\alpha.\alpha)$

leads to a complete and sound unification algorithm, we have to prove that[6]

$$\left(\sqsubseteq_\beta^*;\sqsubseteq^*\right) \supseteq (\sqsubseteq \cup \sqsubseteq_\beta)^*$$

This would mean that if we can find an arbitrary sequence of instance operations $\sqsubseteq$ and β-reductions $\sqsubseteq_\beta$ from some type τ to some other type τ', then we can also find a sequence in which all the $\sqsubseteq_\beta$ steps appear before all the $\sqsubseteq$ steps:

$$\tau\left(\sqsubseteq_\beta^*;\sqsubseteq^*\right)\tau'$$

The proof of this property can be reduced to the proof of commutation of $\sqsubseteq$ and $\sqsubseteq_\beta$:

$$(\sqsubseteq_\beta;\sqsubseteq) = (\sqsubseteq;\sqsubseteq_\beta)$$

This proof is left for future works, but it should rely on the fact that $\sqsubseteq$ doesn't introduce new redexes and that reduction is essentially a rearrangement of sub-graphs.

To summarize, the new unification algorithm performs $\approx$ operations if necessary, where $\approx$ is extended to contain $\sqsubseteq_\beta \cup \sqsupseteq_\beta$, defined in Figure 15. Notice that it already contains $\sqsubseteq_\delta \cup \sqsupseteq_\delta$, defined in Figure 18, which is a special case of inert node unsharing, as explained above. Precisely, the algorithm reduces each β-redex in the sub-graphs to be unified, duplicating the λ-node and its sub-graphs, if it is the child of more than one @-node. This done, it proceeds as before.

See Appendix A for a complete example where, at the end, a sub-graph graph is duplicated and then β-reduced.

[6] where ; is the composition of relations: $R_1; R_2 = \{(a,b)\,|\exists c.a\,R_1\,c \wedge c\,R_2\,b\}$

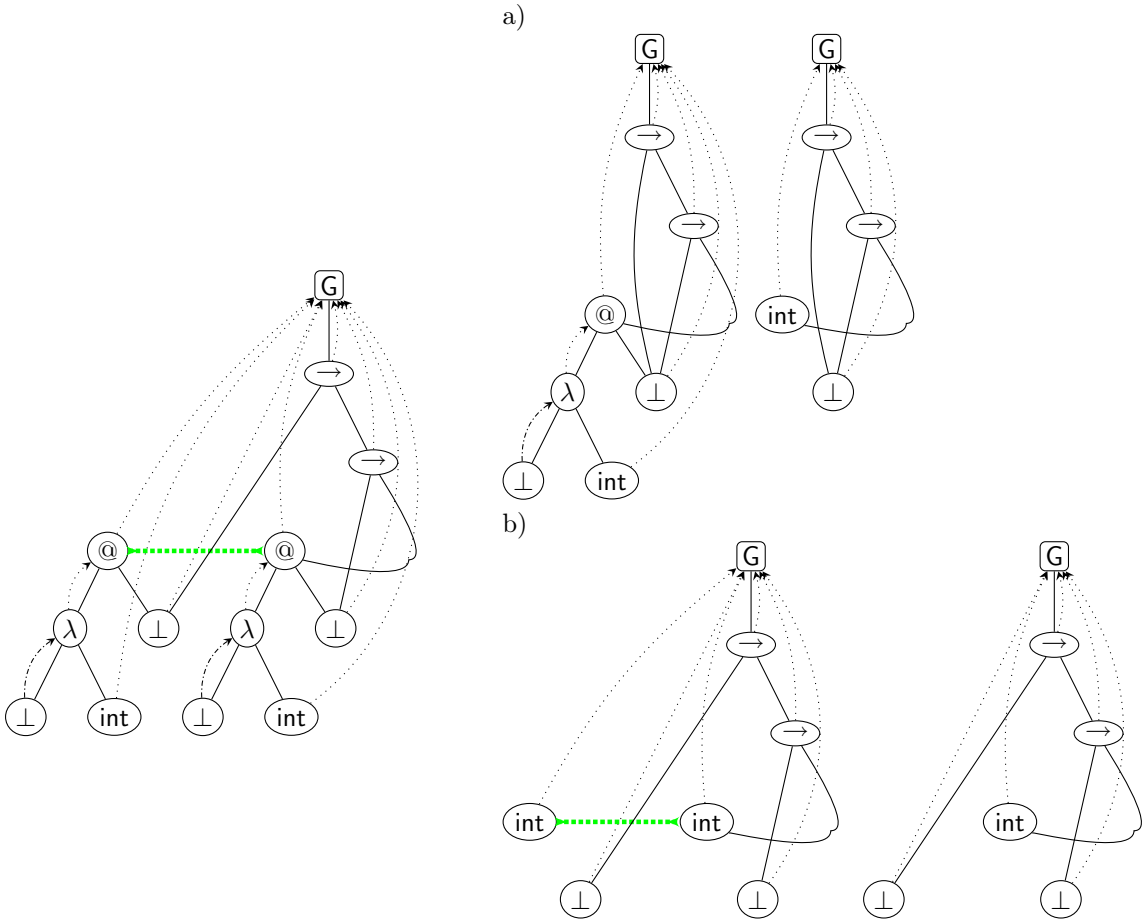


Figure 17: Example of confluence problem

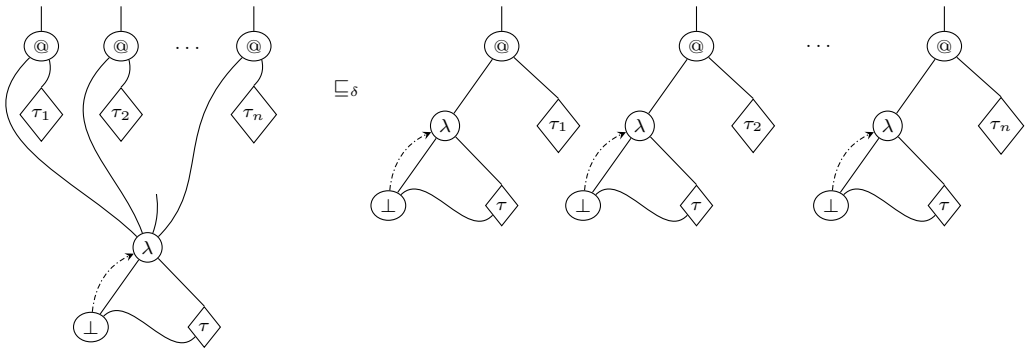


Figure 18: Type equivalence: unsharing of a redex

4 Conclusions

The goals of the research project were essentially achieved in spite of a long process of familiarizing with the vast, non-standard research context. We found a way to combine the implicit polymorphism based on partial type annotation of MLF with the fully explicit world of System F to introduce features of F_ω into MLF, extending its graphical presentation. The result shows that partial type inference in presence of higher-order polymorphism is possible and practical and that is not necessary to reintroduce the need of a huge amount of type annotations as in QML, that leads the formalism to be unusable as a programming language.

This research could be the basis of the type system of a new functional programming language, in which modules and functors are first-class objects. A prototype of this type system is ongoing work. We plan to finish a first version until the end of the Master project.

We listed some interesting properties which should suffice to show correctness of the extension but we didn't carry out the proves. This is left out for future work.

Another interesting improvement left for future work is to perform inference for higher-order type variables based on second order matching.

References

[1] Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, June 2007.

[2] Boris Yakobowski. Types et contraintes graphiques : polymorphisme de second ordre et inférence. PhD thesis, Université Paris 7, December 2008.

[3] Didier Rémy and Boris Yakobowski. From ML to MLF: Graphic type constraints with efficient type inference. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, British Columbia, Canada, pages 63-74. ACM Press, September 2008.

[4] Benjamin C. Pierce. Types and Programming Languages.The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142

[5] Daan Leijen. Flexible types: robust type inference for first-class polymorphism. POPL 2009: 66-77

[6] Claudio V. Russo and Dimitrios Vytiniotis. QML: Explicit First-Class Polymorphism for ML. To appear at ML Workshop 2009.

I'd like to thank Didier for his complete support and constant availability. This work wouldn't have been possible without our in-depth discussions during the whole period and his instant proof reading at the end.

Also, I want to thank the whole Gallium team for their warm reception and their friendly and constructive help, where I learned a lot of things.

Many thanks also to the department at the Università di Pisa, especially to Prof. Ferrari for his administrative support and commitment, without which this project wouldn't have been possible.

A Box

This is a full example which helped us to verify how typing performs for the encoding of a module.

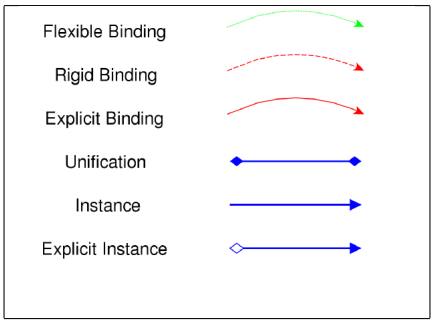
$$
\begin{array}{l}
(\Lambda\,(Box) \\
\quad \lambda\,(box : \forall^{e}\,(\alpha)\,\alpha \to Box\,\alpha) \\
\quad \lambda\,(unbox : \forall^{e}\,(\alpha)\,Box\,\alpha \to \alpha) \\
\qquad unbox\,[\mathsf{int}]\,(box\,[\mathsf{int}]\,5) \\
)\,[\lambda\,(\alpha)\,\forall^{e}\beta\,(\alpha \to \beta) \to \beta] \\
(\Lambda\,(\alpha)\,\lambda\,(x : \alpha)\,\Lambda\,(\beta)\,\lambda\,(f : \alpha \to \beta)\,f\,x) \\
(\Lambda\,(\alpha)\,\lambda\,(b : \forall\,(\beta)\,(\alpha \to \beta) \to \beta)\,b\,[\alpha]\,(\lambda\,(x : \alpha)\,x))
\end{array}
$$

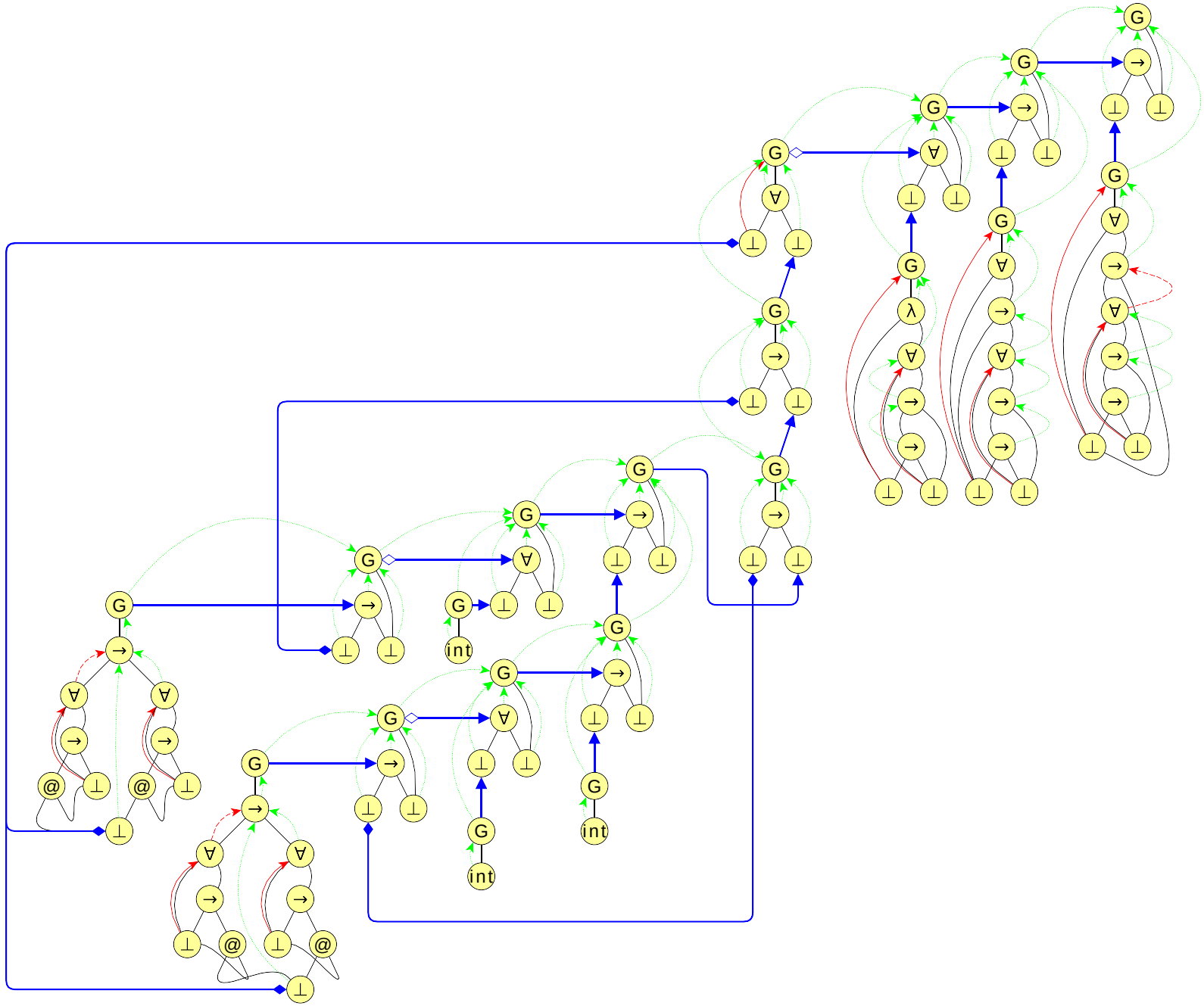
Indeed, the first three lines can be seen as the signature of a module and the last three lines as its implementation, while the fourth line is a line of code which uses the module. The module provides the abstract type operator Box and two functions box and $unbox$. $Box\,t$ represents just an unary tuple, a singleton, of an element of type t. (Like $t_1 * t_2$ represents a binary tuple, a pair, of two elements of types t_1 and t_2) The function box constructs such a box and $unbox$ deconstructs it. For the line of code in the middle, Box is abstract and a box can only be created and used by the functions in the module.

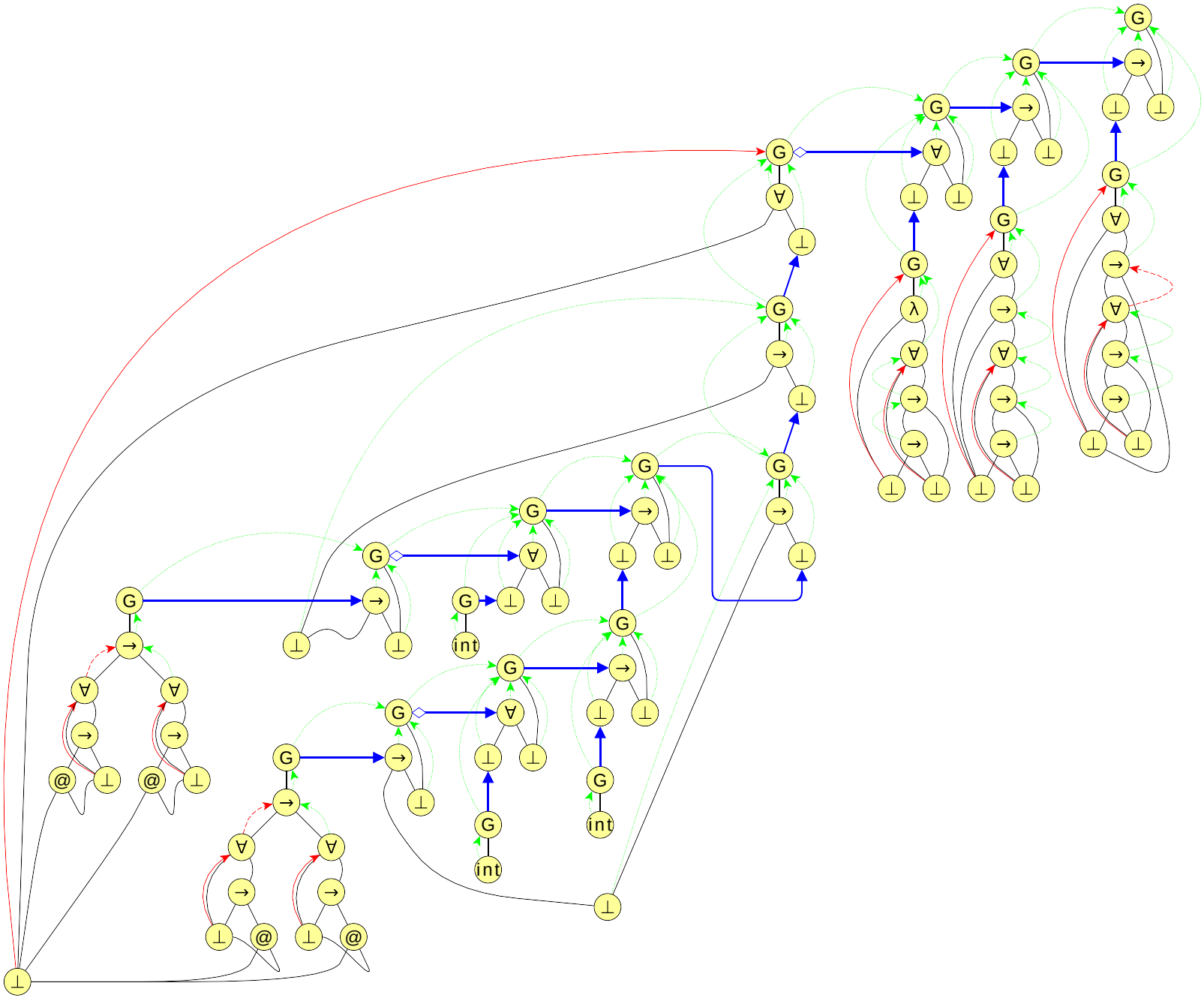
To make it a little more complicated, the implementation of the type is based on the Church-encoding of general tuples. An n-tuple is for Church a function, which takes as argument a function f of type $t_1 \to t_2 \to \ldots \to t_n \to \beta$ and which applies f to all n objects it encapsulates. This way, to use a value at some position k of a tuple, you just have to apply it to a function which return its kth argument. For instance if q is the quadruple (a, b, c, d) then $q\ (\lambda(x)\lambda\,(y)\,\lambda\,(z)\,\lambda\,(w)\,y)$ returns b.

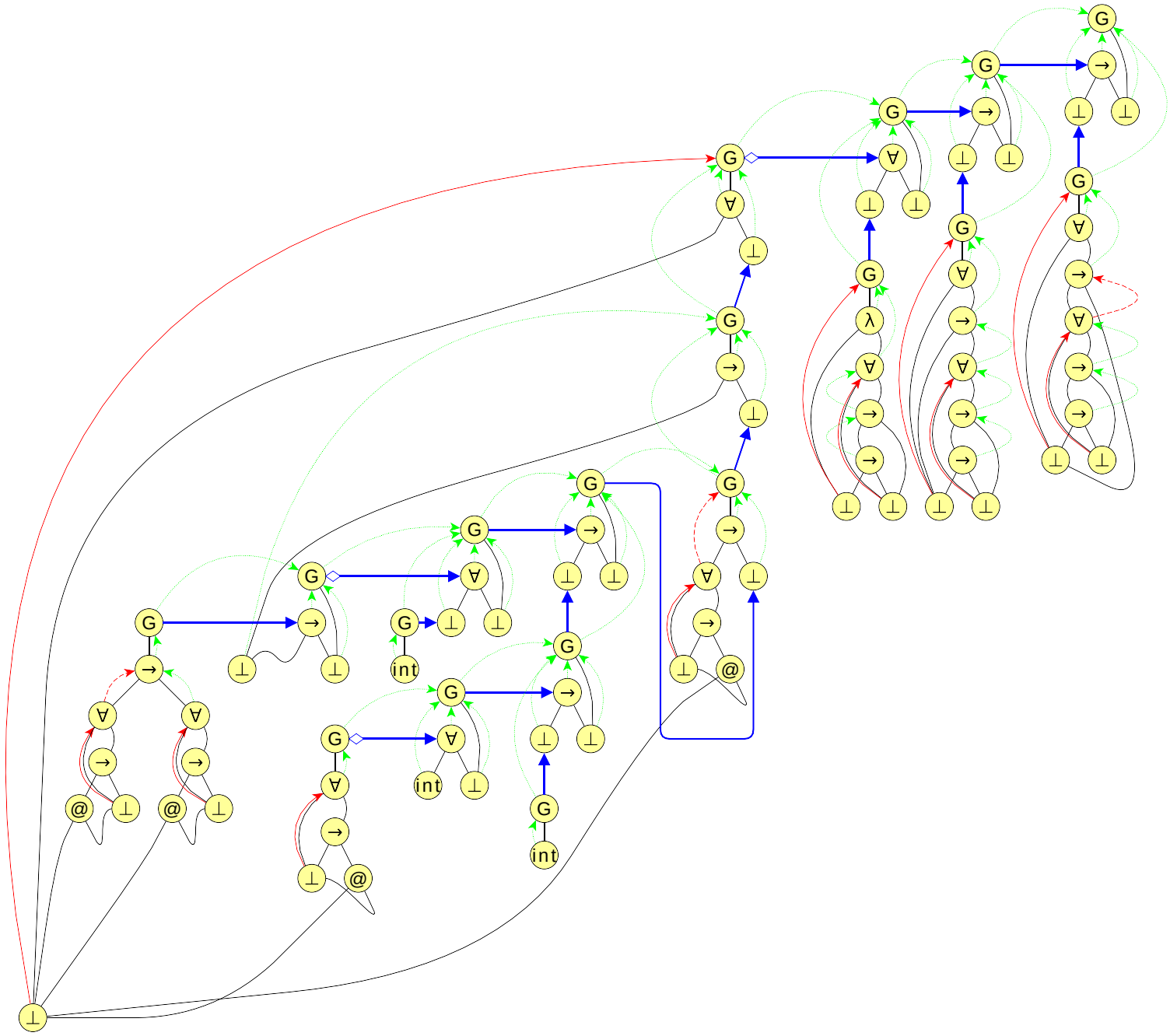
In the case of an 1-tuple, the destructor, here called $unbox$, would apply just the identity to such a value.

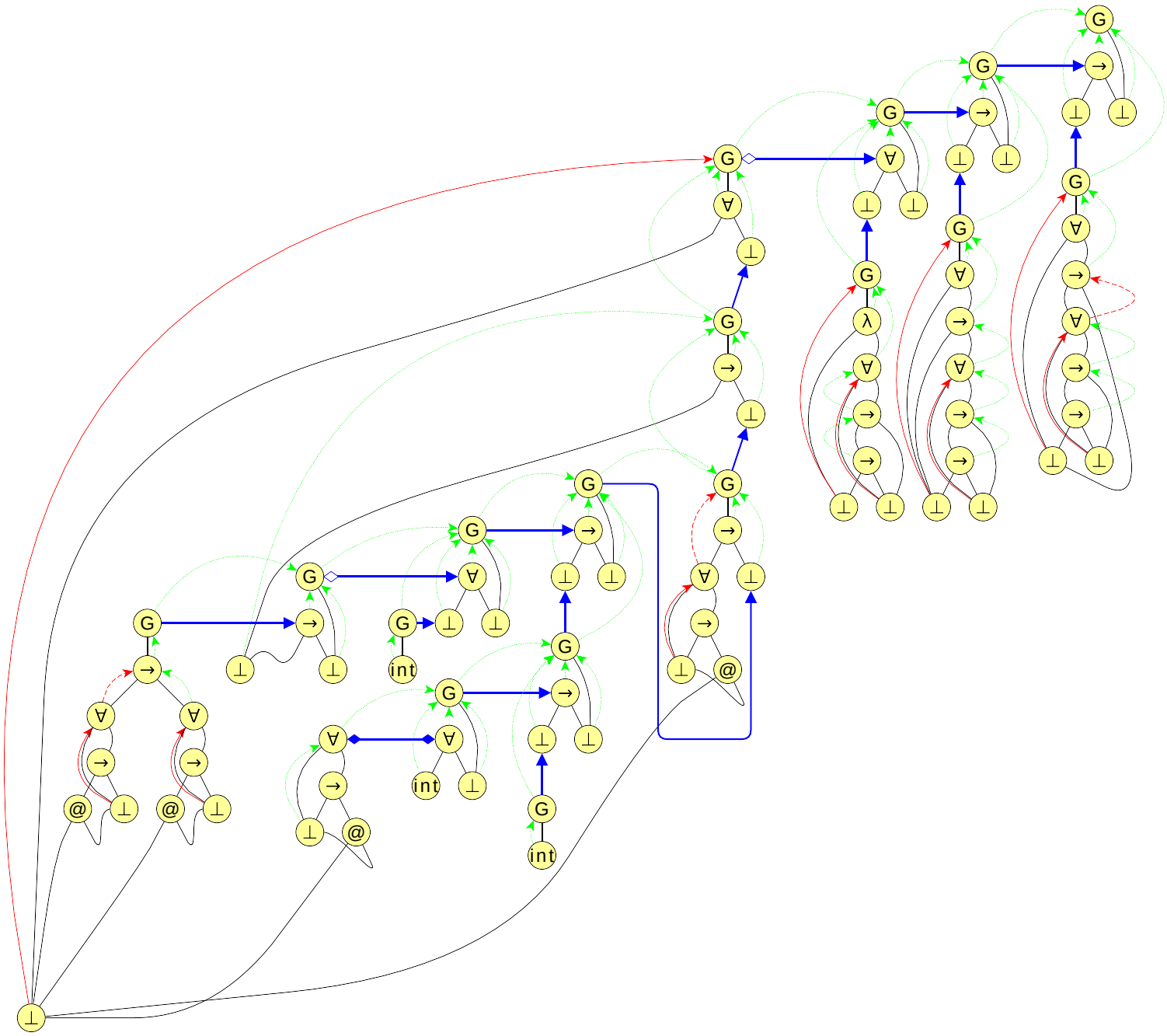
The graphical notation for the example is slightly different, because we used a graphical editor to create it. See the legend below to find the correspondence of the several types of edges.

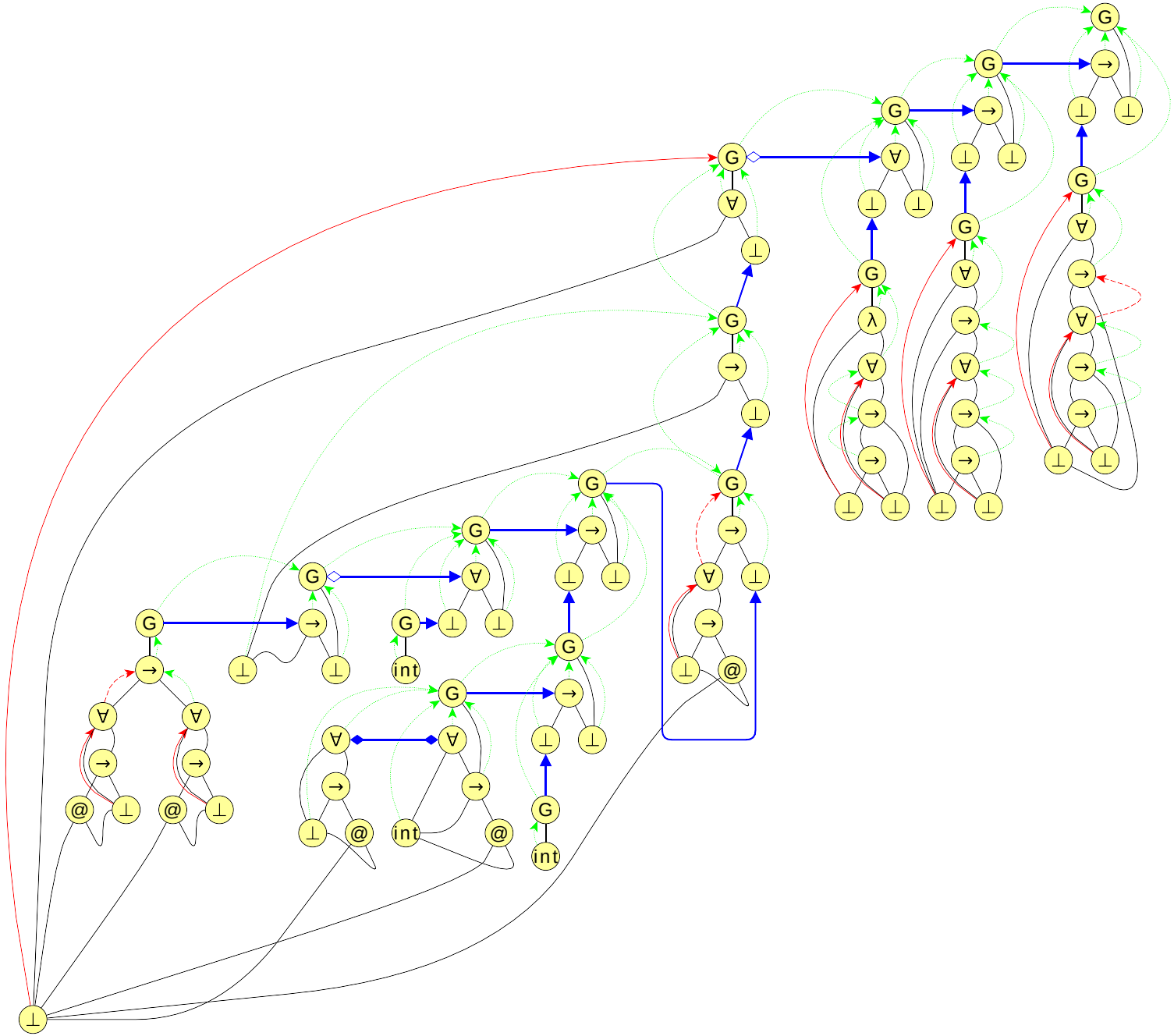


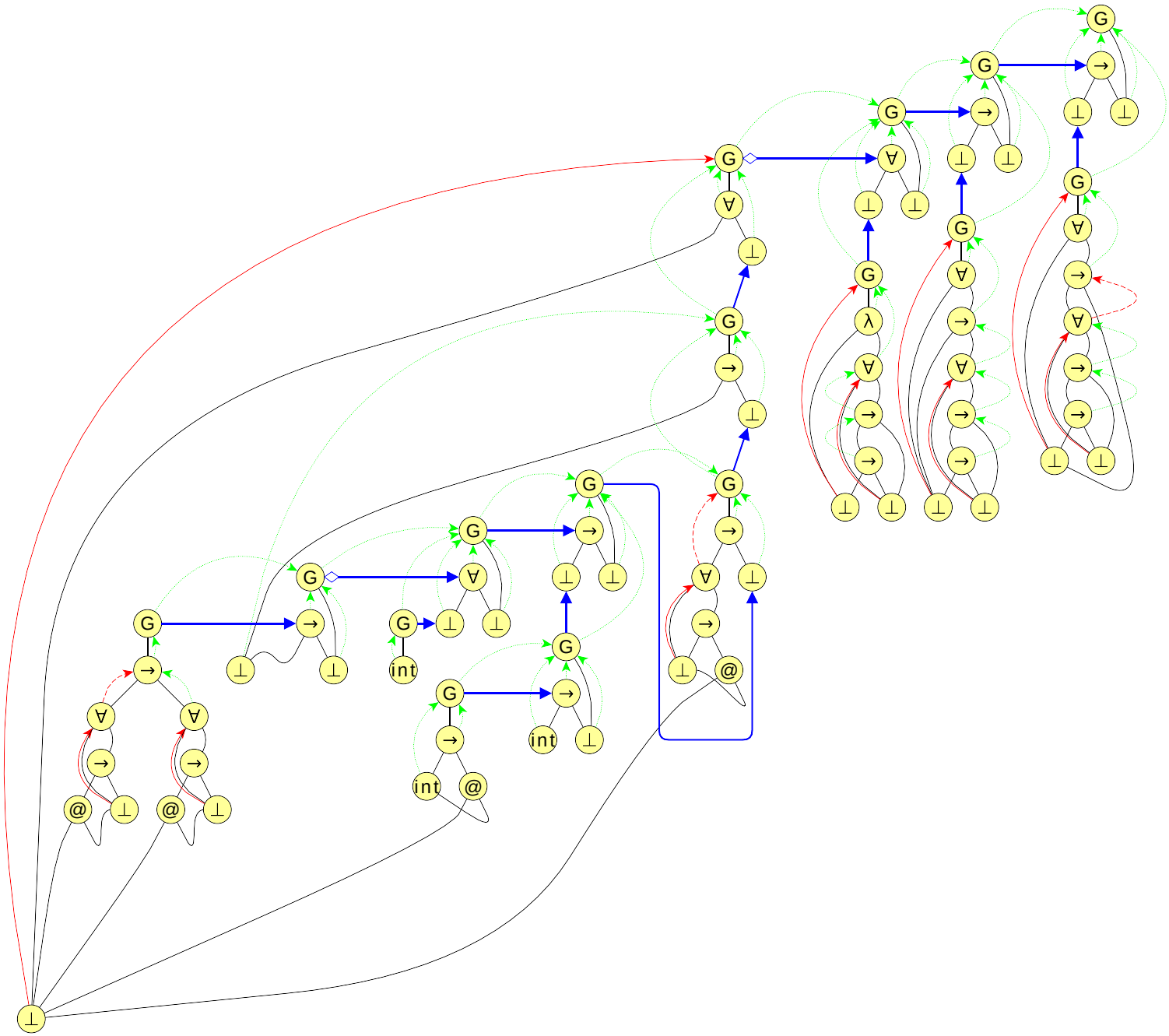

G
∀
⊥
→
λ
@
int

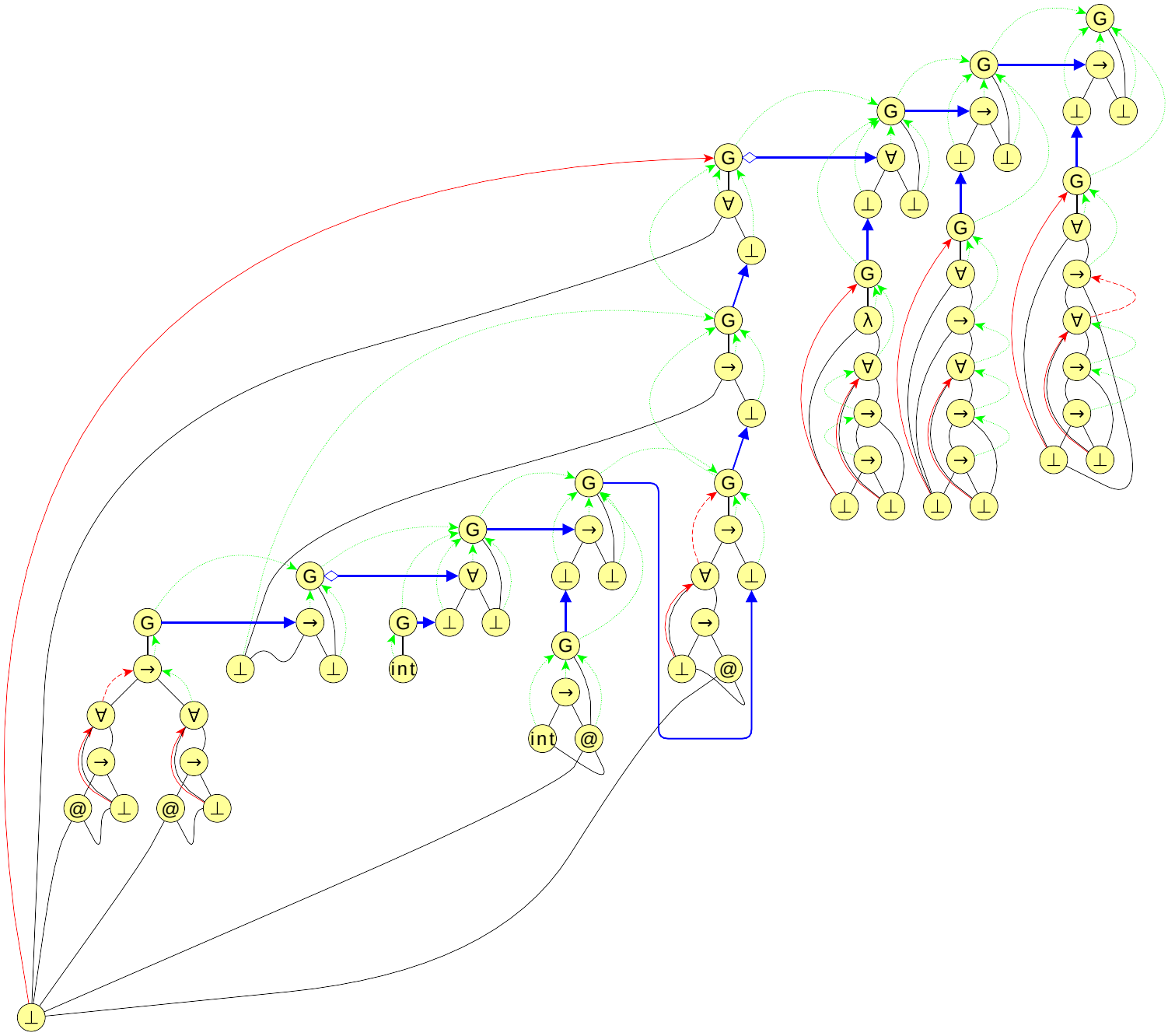

G
∀
⊥
→
λ
@
int

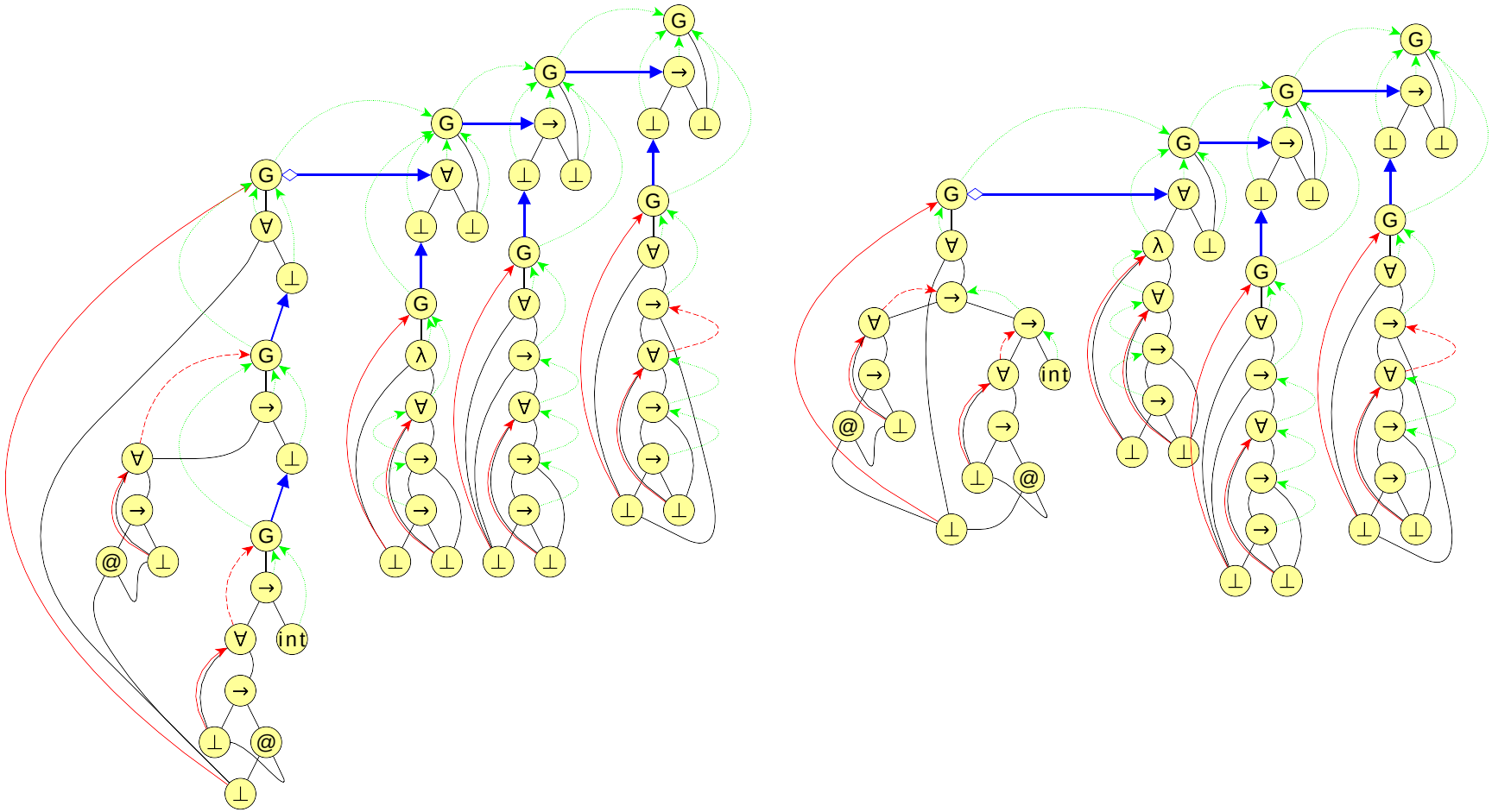

G
∀
⊥
→
λ
@
int

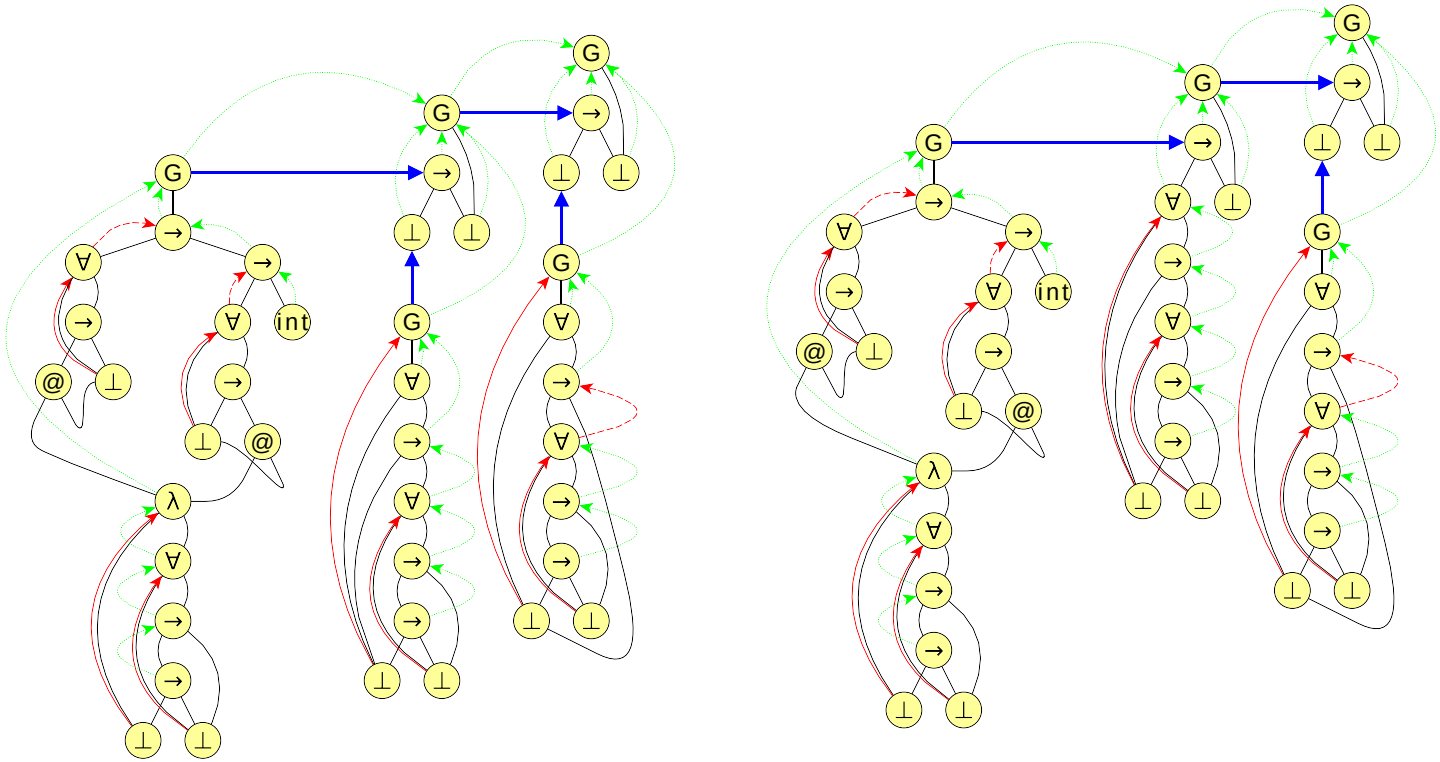

G
∀
⊥
→
λ
@
int

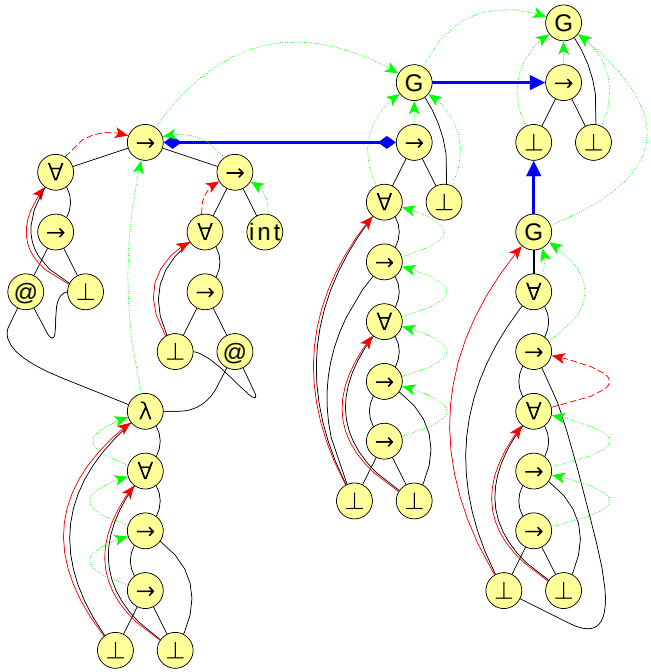

G
∀
⊥
→
λ
@
int

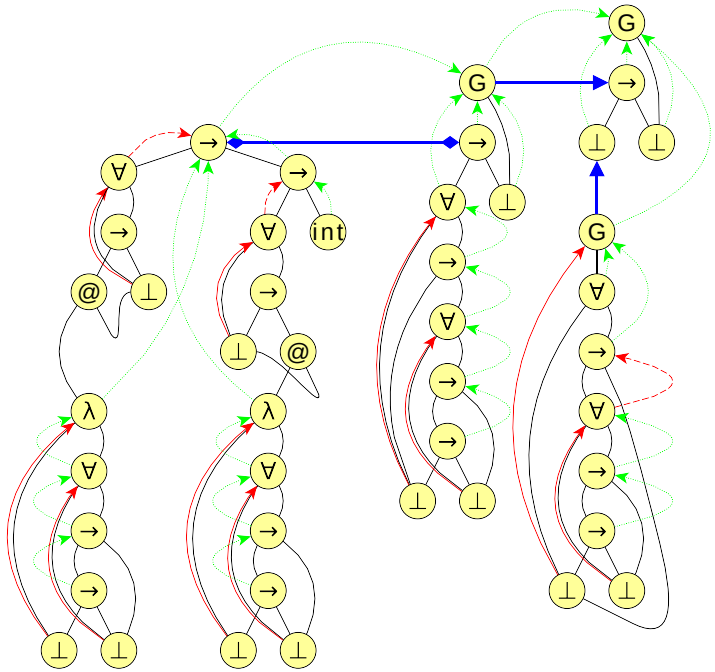

G
∀
⊥
→
λ
@
int

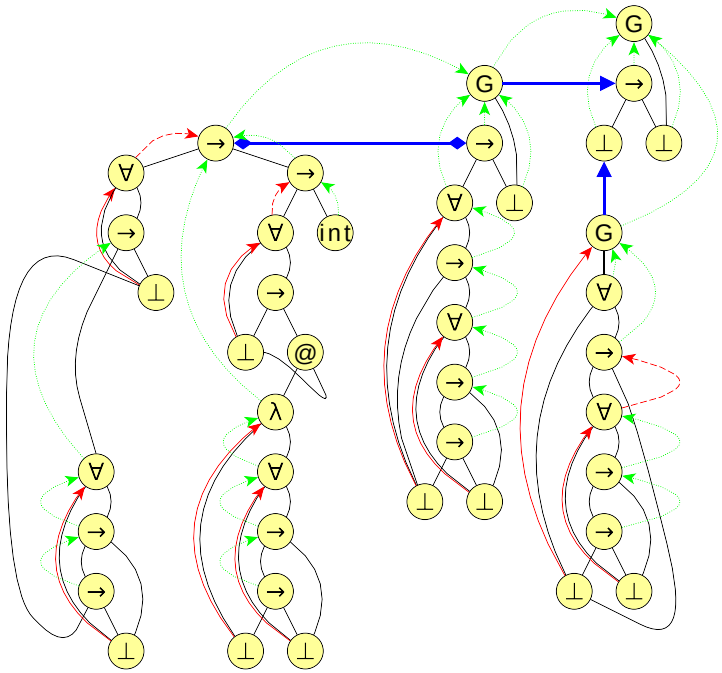

G
∀
⊥
→
@
int
λ

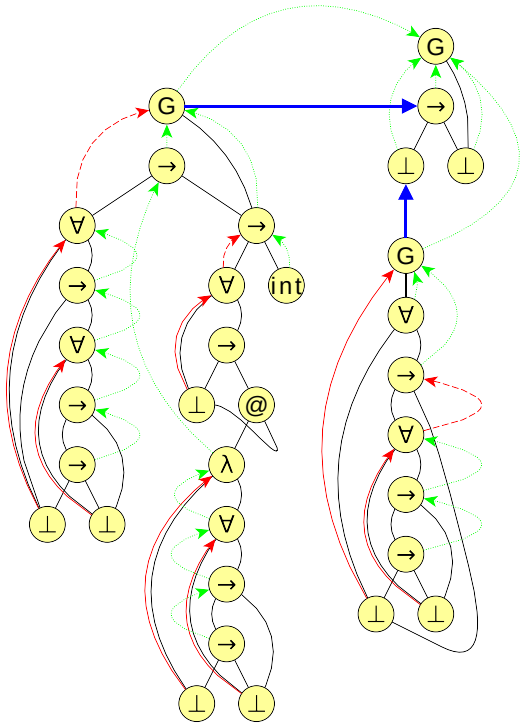

G
∀
⊥
→
@
int
λ

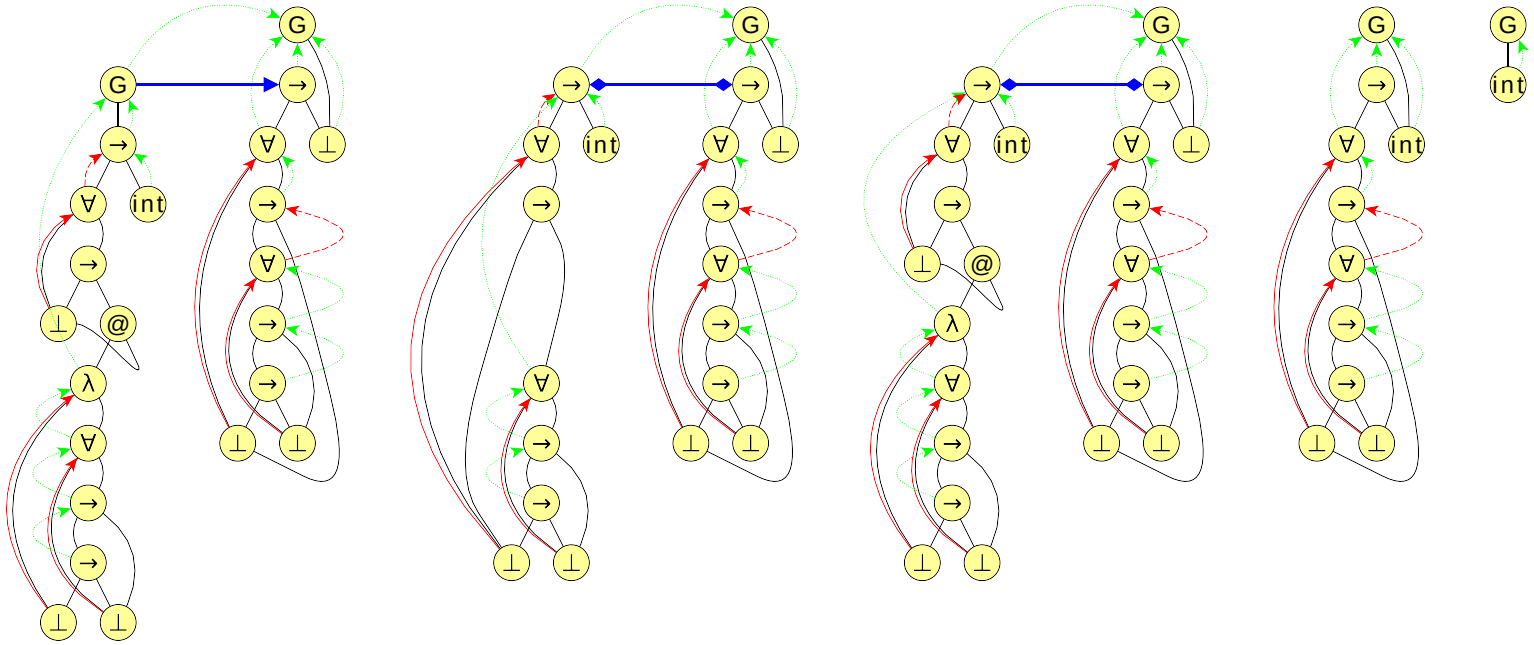

G
→
∀
int
@
⊥
λ


G
→
∀
→
@
⊥
∀
int
→
⊥
@
λ
∀
→
→
⊥
⊥
G
→
∀
⊥
→
∀
→
→
⊥
⊥
G
→
⊥
⊥
∀
→
∀
→
→
⊥
⊥


G
→
∀
→
@
⊥
∀
int
→
⊥
@
λ
∀
→
→
⊥
⊥
G
→
∀
⊥
→
∀
→
→
⊥
⊥
G
→
⊥
⊥
∀
→
∀
→
→
⊥
⊥


G
→
∀
→
⊥
int
@
λ


G
→
∀
int
@
λ


G
G
→
∀
int
→
⊥
@
λ
∀
→
→
⊥
⊥
G
→
∀
⊥
→
∀
→
→
→
⊥
⊥
G
→
∀
int
→
∀
→
⊥
⊥
G
→
∀
int
→
⊥
@
λ
∀
→
→
⊥
⊥
G
→
∀
⊥
→
∀
→
→
→
⊥
⊥
G
→
∀
int
→
∀
→
→
→
⊥
⊥
G
int