筑 波 大 学 大 学 院 修 士 過 程

理 工 学 研 究 科 修 士 論 文

GHC と Java のメッセージ インタフェースに関する研究

飯 塚　和 久

平 成 １１ 年 ２ 月

筑 波 大 学 大 学 院 修 士 過 程

理 工 学 研 究 科 修 士 論 文

GHC と Java のメッセージ インタフェースに関する研究

飯 塚　和 久

主任指導教官 電子・情報工学系　田中 二郎

目次

第1章　はじめに

現在、多種多様なアプリケーションが存在し、それらをインプリメントするためのプログラミング言語も数多く存在する。アプリケーションによっては、複数のプログラミング言語を用いて、記述が行われることもある。

Java は、インターネット普及にともなって、最近注目されている言語である。しかし、Java は GUI やインターネット関係のプログラミングには適しているといわれているが、他のプログラミングの分野に、必ずしも向いているとは限らない。エキスパートシステムなどに代表される知識処理の分野では、論理型言語と呼ばれるクラスの言語が適しているといわれており GHC もこのクラスの言語の 1 つである。

ここで、知識処理分野のアプリケーションで、知識処理の計算を GHC で、GUI 部を Java でといったような、2 つのプログラミング言語にまたがったものを考えてみる。このようなシステムの場合、言語間で何らかの通信を行う必要があり、これらを行う適当なインタフェースが必要になる。

このように、異なったプロラミング言語間でアプリケーションが構成されている場合、2 つの言語のプログラム間で、データをやり取りするための機構が必要である。本研究では、GHC と Java の 2 つのプログラム言語間で、このようなデータ通信のためのインタフェースについて検討する。

今回検討するインタフェースは、オブジェクト指向言語と、並列論理型言語という、全く異なる言語パラダイムを持った、2 つの言語間でのインタフェースである。よって、これらの言語間インタフェースの構成と実装に当たっては、各言語の特性を考慮に入れる必要がある。本稿では、このインタフェースに関して、2 つの言語で利用しやすい、、相手言語の機能を有効に利用するためのインタフェースについて、そのインタフェース形態について検討を行う。また、そのインタフェースの実装を行ったので、その実装システムと、インタフェースの利用例について述べる。

第2章　メッセージインタフェース

2.1 GHC と Java

GHC[1]は、並列論理型言語と呼ばれる言語の 1 つである。Prolog に代表される、いわゆる論理型言語に由来する言語で、その処理を並列に実行させることが出来るように設計された言語である。論理型言語のように知識処理の記述を簡単に行うことができ、さらに、言語レベルでの細かい並列処理の記述が可能であるといった特徴を持つ。

GHC の名前は、"Guarded Horn Clauses"に由来し、GHC 上でのプログラムは、Prolog 等で利用した Horn Clause に、ガードと呼ばれる選択条件を付与した"ガード付きホーン節"を記述するすることによって行われる。ホーン節に付与されたガードは、1 つのゴールを構成する複数のホーン節の中から、1 つの節を選択する条件として利用される。各ホーン節では、ホーン節が選択されたときに展開されるゴールや、変数の内容を同一化するユニフィケーションを記述する。このホーン節中のゴールやユニフィケーションを並列に実行することにより GHC の並列処理が記述される。プログラムは、初期ゴールを与えて、そのゴールをリダクションすることにより実行が行われる。

GHC のプログラミングは、このような節を記述することにより行われるが、これは非常に低レベルな処理であり、実際のプログラムの際には、データ構造やプログラム構造に関して、より高度な概念を用いてプログラムを作成することとなる。

その 1 つとして、“プロセス”という概念がある。あるゴールのリダクションをする過程において、その中で同じゴールを再帰的に呼び出し、その繰り返しが、あるまとまった計算過程とみなせる場合、そのようなものをプロセスと呼ぶ。プロセスは、1 つの機能を持ったエージェントと考えることも出来る。プロセスは、いくつかのゴールをまとめて扱い、プログラムをある程度構造化するための概念である。

GHC のプログラミングでもう 1 つ重要な概念として、“ストリーム”がある。これは、先ほど述べたプロセスをモジュールとして結び付けるためのものである。コンスセルを用いたリスト構造のデータを利用して、複数のプロセスを結びつける。これは、リストのデータ部分にある論理変数を利用してデータをやり取りするのものである。このようなストリーム通信をプロセス間に適用し、様々な機能を持ったプロセスを結び付けることによりプログラムが構築される。

プロセスと、ストリーム通信は GHC のプログラミングでもっとも基本となるものである。この 2 つの概念を基にして、フィルタ、マージャ、サーバ、要求駆動、差分リストなど、さまざまな概念を適用しながプログラミングを行われる。

GHC の代表的な処理系として、GHC のプログラムを一旦 C のプログラムに変換し、一般の UNIX システムで実行できるようにした KLIC[2]などがある。また、今回対象にする GHC は、ガード部分にユーザ定義のゴールを記述しない FGHC と呼ばれる GHC を対象に議論を行う。

Java[3]は、プラットフォームに依存しないプログラムと実行コードを持つインタプリタ形式の言語であることや、GUI や WWW 上のプログラミングが容易に利用可能といった特徴を持つ。また、Java はオブジェクト指向言語の 1 つであり、プログラムの再利用が簡単に行うことができ、大規模なシステムの記述にも適した言語である。

オブジェクト指向言語である Java のプログラムは、すべて“クラス”という形で記述する。クラスの記述は、継承を利用して、親クラスであるスーパークラスの機能を利用した処理の記述と、そのクラスの独自の機能を記述することによって行われる。クラス独自の機能の記述は、オブジェクトの持つ内部属性を表現するためのフィールドと、プログラム処理を記述するメソッドによって行われる。メソッドには、いくつか種類があり、クラス自身に関する処理を行うクラスメソッド、クラスのインスタンスを生成する際に行う処理を記述するコンストラクタ、インスタンスの動作を記述するインスタンスメソッドがある。

プログラムの動作を記述するのは、クラスメソッドや、インスタンスメソッドなどのメソッドである。メソッドは、手続き呼び出しの形をしており、この中で for ループや if-then 分岐などの制御構造を利用し、オブジェクトのフィールドの変更や他のメソッドを呼び出したり、他のオブジェクトに対して操作を行といった処理を記述する。手続き呼び出しであるという点では、C や Pascal 等の手続きや関数に似ているが、異なるのは、その呼び出しの際にクラスやインスタンスを指定して行う点である。メソッド呼び出しは、オブジェクト指向方法論のメッセージに対応し、オブジェクト間でのメソッド呼び出し、つまり、メッセージ交換を行うにことによりプログラムが実行される。

Java のプログラムは、クラスによって定義されたオブジェクトを生成したり、そのメソッドを呼び出すことによって実行される。これは、メソッド呼び出しやフィールドの変更を繰り返す逐次的な処理が基本であるが、スレッドという機能を用いて、その処理を分割させ、複数のプロセスによる並列処理の記述も可能である。

2.2 インタフェースの形態

今回のインタフェースは、結合する言語パラダイムがそれぞれで異なる。同一言語間の通信は、データ構造や、プログラム構成が同一のため、通信の手法についてのみの問題となるが、今回のように 2 つのシステムが異なったものであるため、この差異に着目して考察する必要がある。

まず考えられるのは、双方の言語における言語パラダイムに沿うような利用しやすいインタフェース形態である。2 つの言語が違ったものであるため、その言語に存在しない特別な記述法を使ったり、新たなメカニズムを用意して、それを利用するといった手法も考えられる。しかし、このような場合、インタフェースを利用したプログラムの部分が他の部分と全く違う形となり、プログラムの意味が理解しにくくなる。このため、

インタフェースの形態は、各言語上の普通の記述をすればインタフェースが利用できるといった、自然な形である必要がある。

しかし、このようなインタフェースにすることによって、機能的な制限を与えてしまっては、インタフェースの利用価値が低減する。インタフェースは扱いやすさとともにその機能が重要である。そこで、相手側の言語にのみ存在するような機能でも、もう一方の言語から利用できるようにし、様々な処理が可能になるようなインタフェースが求められる。

また、インタフェースを利用する目的は、自身の言語で提供していない特別な機能を利用するためである。このような機能は、よく使われる機能でもあるため、簡単に利用できる形態が求められる。上記のような様々な場合に適用できるインタフェースだけでなく、より利用しやすさを実現するような、簡単で単純なインタフェースについても検討する必要がある。

最初に挙げた、双方の言語で利用しやすいインタフェースについて検討する。このような言語パラダイムに依存しないインタフェース形態は、ベース言語に依存しないようなメッセージ通信として、他のインタフェースでも実現されている。例えば、UNIX システムでは、プロセル間の通信としてパイプや、シグナル、ソケット通信などが提供されており、これらは、異なる開発言語間で共有できるようになっている。Java と今回 GHC の処理系として利用した KLIC では、ともにソケット通信の機能を提供しており、これは 1 つのインタフェースとなっている。しかし、ソケット通信では、単に文字列をやり取りするだけであり、まとまったデータを送る際など、扱いにくいインタフェースである。

そこで、言語間で同じデータ型を用いてストリーム型通信を行うものとして、データストリームというインタフェースを提案する。これは、GHC 側では、GHC でのプログラミングの基本的概念であるストリームと同様に扱えるといった特徴を持ち、Java 側でも普通のソケットと同様の利用形態で利用できるようにすることで、どちらの言語でも自然な形で利用できるインタフェースとなっている。

上記のストリーム通信は、非常に扱いやすく、かつ各言語で簡単に利用できるインタフェースではあるが、これだけでは、機能的に不充分である。Java について考えると、2.1節で述べたように、Java のプログラムの基本であるオブジェクトや、それを操作するメソッドなどが GHC 側から利用することが出来ない。これでは、GHC との通信で行える処理の範囲が限定される。

そこで、GHC 側からオブジェクトを操作するようなインタフェースが必要である。つまり、オブジェクトを生成し、そのオブジェクトに対してメソッドを呼び出したり、フィールドの値を参照、変更するようなインタフェースである。これらの機能は、当然として GHC 側では存在しない機能である。よって、このインタフェースは新たな機構となるわけであるが、先に述べたように、GHC のプログラミングに即したインタフェ

ース形態が求められる。そこで、今回は、このオブジェクトに対する一連の操作を、GHC のプログラミングに沿ったインタフェースを実装した。これは、オブジェクトをプロセスとみなし[4]、そのプロセスへのメッセージを送るといった形のインタフェースである。

最後に、GHC 側の機能を Java 側から簡単に利用するインタフェースを検討する。GHC のプログラムは先に述べたようにプロセスより成り立つ。よって、データストリームのインタフェースがあれば、GHC のプログラムとの通信は可能である。しかし、GHC のプロセスの中には、非常に機能がまとまっていて、入力データを与えると、それに対して何らかの結果を返すようなものが存在する。簡単な例では、n 番目の素数を求めるとか、ハノイの塔を解くといったものである。これらを Java 側から簡単に利用できるインタフェースが考えてみる。これらは、入力に対する戻り値を得るということで手続き呼び出しに対応することが出来る。このような GHC のプログラムを、1 つの手続きとして扱うことが出来るインタフェースとして、手続き呼び出しのインタフェース[5]を提案する。

2.3 データストリーム

データストリームは、GHC と Java のプログラム間で、データをストリーム通信の形態でやり取りするためのインタフェースである。このインタフェースは、GHC におけるプロセスとストリームに基づくプログラミングに適合し、GHC のプログラムから簡単に利用できるインタフェースとして利用が出来る。また、Java 側での扱いは、ストリーム通信を扱うオブジェクトを生成し、そのオブジェクトに対してデータの読み込みや書きこみのスレッドを呼び出すことで操作が出来るため、双方の言語で扱いやすいインタフェースとなる。

このインタフェースは、UNIX のソケット通信と同じような形態を持つが、単に文字列だけではなく、構造を持ったデータを扱えるようにしている点が異なる。文字列データのみを用いた通信では、複雑なデータを処理するのは難しく、扱いにくいといった問題が生じる。そこで、データストリームのインタフェースでは、文字列以外のデータ型を扱えるようにし、さらに、複数のデータを同時に処理するために構造化データを扱えるようにすることによって、プログラム間での通信を簡単化させることが可能である。

今回は、2 つの異なった言語間での通信であり、それぞれの言語で扱えるデータ型も異なる。よって、データストリームのインタフェースで用いるデータ型は、この点を考慮する必要がある。この問題の 1 つの解決法は、双方の言語で扱える共通のデータ型を利用する方法であり、もう 1 つは、相手側の言語で利用できるデータ型をもう一方の言語でも利用できるようにすることによって、どちらの言語のデータ型も利用できるようにしておき、双方の言語のデータが利用できるようにする方法である。

Java はオブジェクト指向言語であり、抽象データ型を採用しているため、基本的なデータ型の他に、様々なデータの形を作成、利用することが出来る。このような Java のデータを GHC 側で扱うには複雑な処理が必要であり、簡単で容易に利用できるようなインタフェースとして提案したデータストリームのインタフェースの目的に外れる。そこで、Java と GHC で共通なデータ型を与え、これらを介してデータを扱うようにしたほうがよい。

共通なデータ型として、ここでは、GHC のデータ型を用いる。GHC のデータ型は、基本的なデータ型とともに、ファンクタという構造型データを扱えるため、複雑なデータを与えることも可能である。また、GHC 側での扱いが簡単であり、Java 側でも、GHC のデータ型を扱うクラスという形で利用できるため、特に複雑な処理をしなくても、双方の言語で簡単に利用できるデータとして扱うことが可能である。

ここで挙げているデータストリームのインタフェースの「ストリーム」と、先に述べた GHC のプログラミングに用いられる「ストリーム」とは異なるものである。GHC では、ストリームに渡すことが出来るデータとして、一般的なデータの他に未完成メッセージと呼ばれるデータを渡すことが出来る。これは、データとして、変数の参照を渡すことにより、ストリームの相手先が受け取ったデータを元に、戻り値などをストリームの送り先が受け取ることが出来る。GHC のプログラミングでは、この機能を用いて 1 つのストリームで双方向通信が可能となっている。このようなストリームは、GHC のプロセス間の通信では便利であるが、Java のプログラムと GHC のプロセス間で通信する場合は、このようなインタフェースは扱いにくいものとなる。これは、Java 側で一般のデータと未完成メッセージを区別する必要があり、インタフェースが複雑なものとなるからである。データストリームのインタフェースは、GHC と Java で簡単に利用できるインタフェースとして考えたものであり、未完成メッセージを含むデータを扱わないストリームを用いることとした。なお、データストリームでは、方向性を持った 2 つのストリームを用いることにより、GHC と Java 間の双方向通信を可能にしている。

2.4 オブジェクト操作

オブジェクト操作のインタフェースは、GHC から Java のオブジェクトを操作するためのインタフェースである。前に挙げたデータストリームでは簡単なデータをやり取りするには便利であるが、複雑な処理が出来なかった。そこで、GHC から Java のオブジェクトを直接扱えるようにすることで、Java のプログラムに対する直接的なインタフェースを提供し、様々なアプリケーションで利用できるようにする。

ここでいう、オブジェクトの操作とは、Java のオブジェクトに対して、オブジェクトを新たに生成し、そのオブジェクトに対して、メソッドを呼び出したり、フィールド

の参照や変更を行うことをいう。これらの操作を GHC 側から行うことにより、Java のプログラムとの通信が行うことが出来る。

2.2節でも述べたように、GHC 側のオブジェクト操作のインタフェース形態は、各オブジェクトをプロセスとして扱い、それらに対してストリームを通じてメッセージを送ることによって、オブジェクトを操作する。このようなインタフェースで利用できるのは、オブジェクトの生成から、そのオブジェクトへの操作、オブジェクトの参照を捨てるといった一連の流れが、プロセスの動作と対応するためである。そこで、Java オブジェクトへのインタフェースとして、オブジェクトに対応するプロセス生成し、このプロセスに対して、GHC のストリームを用いて操作を行うといったインタフェース形態を利用する。

まず、オブジェクトを生成する場合には、オブジェクトの元となるクラス名と、必要に応じて引数を与えることにより、プロセスを生成する。このとき、Java 側では、指定されたクラスのオブジェクトを生成する。

メソッドの呼び出しは、生成されたプロセスに、メッセージを送ることによってメソッド呼び出しを行う。Java におけるメソッド呼び出しは手続き呼び出しであるが、GHC には手続き呼び出しというものは存在しない。そのかわりに、プロセスへのメッセージを与えると、メソッドを呼び出す。メソッドの戻り値は、メッセージに付与された変数へに具体化するという、GHC の不完全メッセージを利用し、メソッド呼び出しのインタフェースを提供する。プロセスへのメッセージは、例えば `call("getItem", ["Top", 2], Ret)` のようになり、"call"というのが、メソッド呼び出しを示し、このファンクタの第 1 引数の"getItem"はオブジェクトが持っているメソッド名を指定していて、次の引数`"["Top", 2]"`で可変個の引数を指定している。そして、最後の引数に、メソッドの戻り値を返す変数を指定していており、メソッド呼び出しの戻り値がその変数に具体化される。

フィールドの参照や変更も、同様にプロセスにメッセージを送ることによって、オブジェクトに対する操作を行う。例えば、`set("name", "setData")` や `get("name", Ret)` といったメッセージによってオブジェクトへアクセスすることになる。

これらのメッセージは、すべてオブジェクトに対応するプロセスに送られる。これらのプロセスは送られたメッセージを元にして、Java のオブジェクトに対して具体的な操作を行う。メソッドの呼び出しや、フィールドの参照の場合は、それらの値を Java 側から受け取り、それぞれの変数へ具体化する。

このプロセスと接続するストリームが閉じられた場合は、オブジェクトの操作を終了したとみなし、このプロセスを終了し、Java 側でのこのオブジェクトへの参照を失う。

Java のプログラムには、単にオブジェクトだけでなく、クラスの持つ静的なメソッドやフィールドが存在する。これらは、そのクラスのオブジェクトを管理したり、オブ

ジェクトに依存しない情報を操作、参照したりすることが出来る。これらをGHCで扱うためのインタフェースも、これらメソッドやフィールドを、静的なオブジェクトに対する操作と考えることにより、同様のインタフェースで扱うことが出来る。

2.5 手続き呼び出し

手続き呼び出しのインタフェースは、GHCのプログラムを、Java側から簡単に利用することが出来るようにするためのインタフェースである。データストリームのインタフェースでは、GHC側で用意されたプロセスに対して、ストリーム通信でデータをやり取りすることが出来るが、Java側から任意のプロセスに対して操作することが出来なかった。手続き呼び出しのインタフェースは、このようなGHCの基本的な概念であるプロセスを、Java側から任意に指定して生成、利用するためのものである。

GHCのプロセスに対しての操作は、すべてストリームを通じて行われる。つまり、Java側でプロセス生成して、何らかの処理をしたい場合、このプロセスのストリームに必要なデータを渡す必要がある。しかし、このようなストリームは複数存在し、また、GHC特有の複雑なデータを処理する必要があるので、このままではJavaから簡単に利用することが出来ない。そこで、プロセスに対する操作を、データ入力のストリームと出力のストリームとを分けて、データを入力すると出力データが決まるといったプロセスに着目し、このようなプロセスをJava側から扱うこととする。

このようなプロセスの処理手順は、入力に対して出力が決まるので、手続き呼び出しの処理の流れと一致する。よって、Java側のインタフェース形態として、メソッド呼び出しの形をそのまま利用することが出来る。これにより、GHCのプログラムをJava側から簡単に利用することが可能となる。

このインタフェースを利用するには、例えば、`jk.call("hanoi", data)`のようなメソッド呼び出しをすることになる。ここで、“`jk`”はインタフェースを提供するオブジェクトで、“`call`”はそのオブジェクトのメソッド名である。メソッドの第1引数としてプロセス名を指定し、“`data`”は、そのプロセスのストリームに与える入力データのストリームのデータを指定する。このメソッドが呼び出されると、GHC側のプロセスが生成されて入力データに従って処理を行う。このプロセスは、入力データと出力データを別々に与えるプロセスであり、処理が終了した時点でプロセスが出力データを出力ストリームに書き出す。Java側で、この値を取得し、これを先のメソッド呼び出しの戻り値として返すことにより、Javaのプログラムから、プロセス実行を伴ったGHCのプログラムとの通信が実現出来る。

第3章　実装

3.1 構成

今回実装を行った環境は、GHC の実行環境として KLIC を用い、Java は JDK1.1 を利用している。このインタフェースは、各言語上でライブラリとして実装され、インタフェースを利用する場合は、このライブラリを用いて通信を行うこととなる。また、各言語上のインタフェース間で通信を行うために、UNIX のストリーム型ソケットを用いている。このソケットをライブラリに指定することにより、通信を行うアプリケーションが決定される。システムの全体的な構成は図 1のようになる。

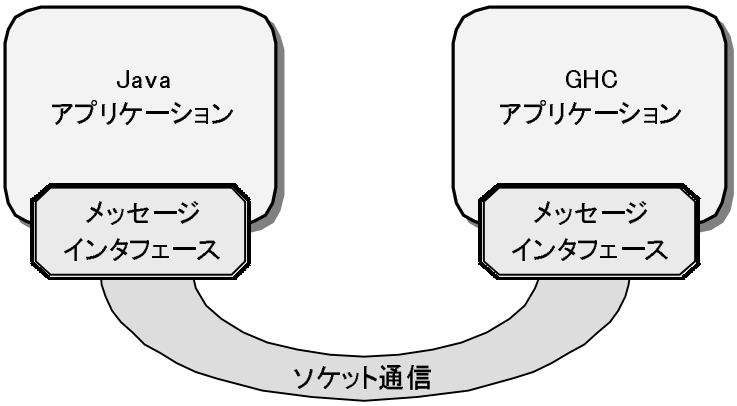

図1 全体の構成

GHC 側におけるメッセージインタフェースの実装は、KLIC のモジュールとして実現されている。インタフェースを利用する場合には、このモジュールのゴールを呼び出すことで使用できる。モジュールは、jk, jkmain, jkinput, jkspool, jkparse, jkwrapstr, jkinspector の 7 つからなる。各モジュールが提供する機能の概略を図 2に示す。ユーザがインタフェースを利用する場合は、jk モジュールの jk/2 というゴールを呼び出す。このゴールが、その他のモジュールを利用して、必要なプロセスを生成し、インタフェースの機能を実現する。

モジュール	機能
jk	必要なプロセスを生成し、メッセージインタフェースの機能を提供する
jkmain	GHC側からのメッセージを受け取り、Java側に渡す
jkinput	Java側からのメッセージを受け取り、処理を行う
jkspool	手続き呼び出しや、ストリーム通信などのデータを保持する
jkparse	Java側から送られた文字列データを解析しKLICのデータを生成する
jkwrapstr	Java側へのデータ送信のための文字列を生成する
jkinspector	インタフェースの内部動作を外部から参照するためのライブラリ

図2 GHC 側のの各モジュールの機能

GHC 側のメッセージインタフェースの内部は図 3に示すように、main, input, spool の３つのプロセスから構成される。main プロセスは、GHC アプリケーションからのメッセージを受け取り、そのデータを Java 側へ送信する。input プロセスは、Java 側から送られたメッセージを受け取り、そのメッセージに応じた処理を行う。spool は手続き呼び出しやデータストリームのデータ等の値を保持する。これらのプロセスはストリームにより接続され、相互にデータをやり取りする。

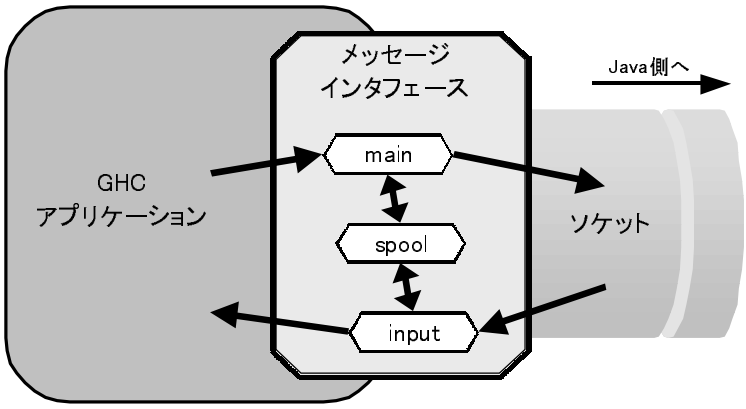

図3 GHC 側の内部構成

Java 側におけるメッセージインタフェースは、いくつかのクラスを用いて実装されている。インタフェースを利用するには、これらのクラスのオブジェクトを利用することになる。主なクラスとして、JK, ReadThread, ArgsManager, Stream, ObjectTable, Data がある。これらクラスの機能の概略を図 4示す。ユーザは、JK クラスのオブジェ

クトを生成し、このオブジェクトのメソッドを呼び出すことによりインタフェースが利用できる。このオブジェクトが、必要なクラスを利用してインタフェースの機能を実現する。また、通信の際に利用される GHC のデータ型を扱う場合には、Data クラスを用いる。

クラス	機能
JK	Javaからのメッセージを受け取りGHC側へ送信する
ReadThread	GHC側から送られたメッセージを受け取り処理する
ArgsManager	手続き呼び出しの戻り値の設定先を管理する
Stream	データストリームのデータ処理
ObjectTable	GHC側から生成されたオブジェクトを管理する
Data	GHCのデータを扱う

図4 Java 側の主なクラスの機能

Java 側のメッセージインタフェースの内部は図 5に示すようになっている。JK クラスのオブジェクトは、Java アプリケーションからのメッセージを受け取り、そのデータを GHC 側へ送信する。ReadThread クラスのオブジェクトは、Java アプリケーションとは別のスレッドで動作し、GHC 側から送られたメッセージを読み取り、メッセージに応じた処理を行う。ArgsManager, Stream の各クラスのオブジェクトは、手続き呼び出しやデータストリームのデータ等を保持する。ObjectTable は、GHC 側から作成されたオブジェクトを管理する。

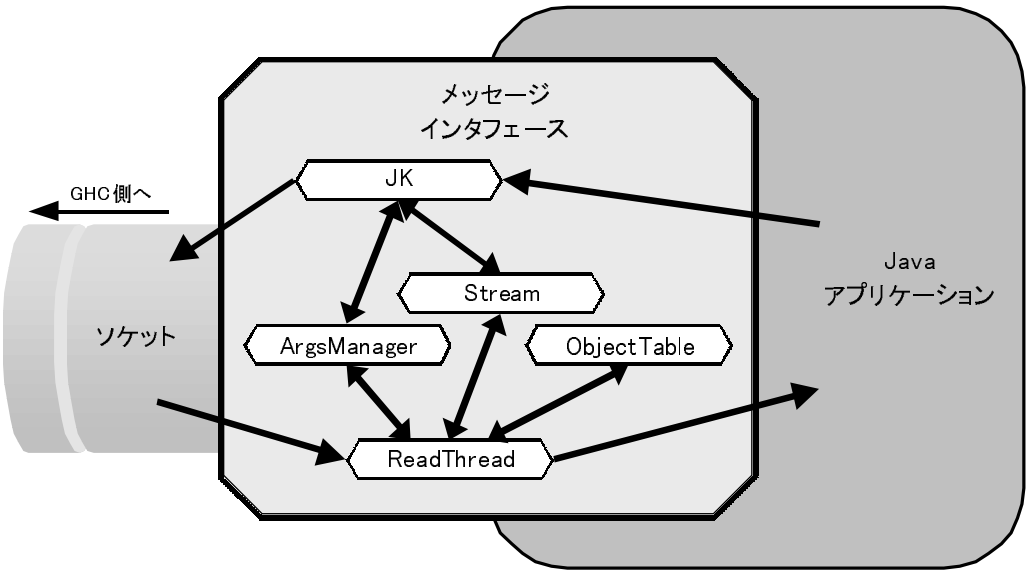

図5 Java 側の内部構成

3.2 データストリームの実装

GHC 側で、データストリームのインタフェースを利用する場合の処理の流れについて述べる。まず、データを送信したい場合は、メッセージインタフェースに対して、

put(送信するデータ)

の形式でメッセージを渡す。“送信するデータ”にはデータストリームを用いて送信するデータを指定する。

このメッセージを main プロセスが受け取ると、まず、データを3.4節で述べる形式でソケットに書きこむ文字列を生成し、Java 側に文字列に変換されたメッセージを送る。このデータを送ると、main プロセスは、インタフェースの処理に戻る。これにより、データストリームで送られるデータは、指定された順序を保存したまま Java 側に渡される。

Java 側では、ReadThread のオブジェクトが、メッセージをソケットより読み込む。ソケットに文字列に変換されていたデータは、Data クラスのオブジェクトに変換され Java で扱える形式となる。そして、メッセージインタフェースを作成したときに生成された Stream オブジェクトにデータが追加される。Stream オブジェクトは、受け取り要求のメソッドが呼ばれていた場合は、その中の最初のメソッドに対して受け取った値を返し、受け取り要求のメソッドのスレッドを再開させる。また、受け取り要求がなかった場合は、そのデータをキューとして保存する。

データを受け取りたい場合は、メッセージインタフェースに対して、

get(受け取るデータ)

の形式でメッセージを渡す。“受け取るデータ”には、受け取ったデータを設定する変数を指定する。この変数には、データを受け取ると、

normal(受け取ったデータ)

の形でデータが設定される。また、データストリームが閉じられた場合には、

abnormal(abnormal)

といったデータが渡される。

このメッセージを main プロセスが受け取ると、データの設定先の変数を spool プロセスに渡す。spool プロセスは、既に Java 側からデータが送られていた場合は、そのデータのうち最初に受け取ったデータを返して、spool に保持していたそのデータを捨てる。データがまだ送られていない場合は、“受け取るデータ”に指定された変数には、その時点ではなにも設定されずに、データが送られてくるのを待つため、設定先の変数をキューとして spool プロセスが保持する。

Java 側で、データストリームのインタフェースを利用する場合は、メッセージインタフェースを実現する JK クラスのオブジェクトに対して、

void put(Data d)

メソッドを呼び出す。“d”には送信するデータを、GHC のデータ型を表す Data クラスのオブジェクトで指定する。このメソッドが呼ばれると、GHC 側の処理と同様にソケットに書きこむための文字列形式にデータを変換してメッセージを送信する。

GHC では、このメッセージを input プロセスが受け取る。そして、文字列で指定されたデータは、GHC のデータ型に変換され、spool プロセスに渡される。spool プロセスは、受け取り要求がすでにあった場合は、そのうち最初の変数に対してそのデータを設定する。また、受け取り要求がなかった場合は、そのデータを spool プロセスがキューとして保持する。

データを受け取りたい場合は、JK クラスのオブジェクトに対して、

Data get()

メソッドを呼び出す。このメソッドは、データを受け取ると Data クラスのオブジェクトとして返す。また、データストリームが閉じられた場合には、null を返す。

このメソッドを呼び出すと、Stream クラスのオブジェクトにデータを要求する。Stream クラスのオブジェクトは、既に GHC 側から送られたデータがあった場合には、そのデータのうち最初に受け取ったデータをメソッドの戻り値として返す。また、データがなかった場合は、呼び出したメソッドのスレッドを一時停止させ、データが送られるのを待つ。

3.3 オブジェクト操作の実装

オブジェクトの生成に際の処理の流れについて述べる。GHC 側でメッセージインタフェースに対して、

instance(クラス名, 引数のリスト表現, オブジェクトプロセスへのストリーム)

の形でメッセージを渡すことにより、オブジェクト生成のインタフェースを利用する。ここで、“クラス名”は、生成するオブジェクトのクラス名である。“引数のリスト表現”は作成するクラスのコンストラクタに渡す引数であり、引数がない場合は、ヌルリスト[]を与える。“オブジェクトプロセスへのストリーム”には、生成されたオブジェクトを扱うプロセスへのストリームを返すための変数を指定する。この変数には、正しくオブジェクトが生成された場合には、

normal(ストリーム)

の形で、生成されたオブジェクトを扱うプロセスへのストリームを返す。また、正しくオブジェクト生成が出来なかった場合は、

abnormal(エラーメッセージ)

の形式でエラーを渡す。

インタフェースでは、このメッセージを main プロセスが受け取る。main プロセスは、オブジェクト識別のためのタグや、エラーメッセージを受け取るための戻り値の識別タグを、spool プロセスに要求し、戻り値を設定する変数を渡す。そして、この戻り値識別のためのタグを、クラス名と引数データとともに Java 側に渡す。また、戻り値が設定されたときの処理を行うプロセスを新たに生成する。このプロセスはオブジェクトの生成を待ち、main プロセスは、メッセージインタフェースの処理に戻る。

Java 側では、ReadThread クラスのオブジェクトがこのメッセージを受け取る。ここで、指定されたオブジェクトを生成するのであるが、インスタンスの作成の際に利用するコンストラクタの呼び出しは、処理が複雑で、長い時間を必要とする場合がある。ReadThread のスレッド中で処理すると、この処理を行っている間は他のインタフェースが利用できなくなり、効率的な作業が出来なくなる。このため、ReadThread のオブジェクトは、新たなスレッドを作成し、ReadThread の処理とは別々に動作させ、他のインタフェースの処理を並列に行えるようにした。

オブジェクトの生成は、まず、java.lang.Class クラスの forName(String)メソッドより、GHC 側から送られた文字列よりクラスを特定し、Class クラスのオブジェクトを得る。オブジェクトを生成する際のコンストラクタは、引数の型によって複数存在するため、指定された引数を元にコンストラクタを探す。ここで、GHC から引数として与えることが出来るデータは、GHC のデータ型を表す Data クラスのオブジェクトしかない。そこで、リスト形式で与えられた引数を、配列型に変換しその引数の数を得る。その引数の数だけ Data クラスを引数として持つコンストラクタを、Class クラスの getConstructor(Class[]) メソッドを用いて取得する。このようにして得られた java.lang.reflect.Constructor クラスのオブジェクトに対して、newInstance(Object[]) メソッドを用いて新しいオブジェクトを生成する。

GHC 側から、コンストラクタの引数として与えることが出来るのは、Data クラスのデータを 0 個以上持つものだけに限られる。しかし、このままでは、Data クラスから、必要なデータを取得する処理が必要になるなど、Java のプログラミングが面倒になる場合がある。そこで、String クラスのオブジェクトや、int 型のデータを引数として持つようなコンストラクタを利用できるようにし、Java のプログラミングを容易にし、GHC 側から利用しやすいインタフェースを実現した。これは、コンストラクタを探す場合に、まずは、上述のように、Data クラスのオブジェクトを引数として持つものを検索する。そこで、対応するコンストラクタが見つからなかった場合には、実引数として与えられたデータが文字列であった場合には、コンストラクタの引数を Data クラスから String クラスに変更して検索する。同様に、引数に整数が与えられた場合には、

int 型として検索を行う。これらの変換は、必要に応じて行われ、Data クラスの引数のまま適用できる場合には、この変換は行われない。具体的な対応の例を図 6に示す。

これらのコンストラクタの検索は、作成するクラスに対応する Class オブジェクトの getConstructors()メソッドを用いてコンストラクタを取得し、この中から適合する Constructor オブジェクトを探す。このようにすることで、頻繁に利用される、文字列や整数引数を持つコンストラクタの利用の利便性を持たせたせることができた。

クラスの持つコンストラクタ	与える引数	呼び出されるコンストラクタ
Test() Test(Data) Test(int) Test(String, int) Test(String, String)	[]	Test()
	[abc]	Test(Data)
	[123]	Test(Data)
	["jk", 123]	Test(String, int)
	["first", "second"]	Test(String, String)

図6 引数と呼び出されるコンストラクタの対応

以上のようにして生成されたオブジェクトは、以降のオブジェクト操作のために、ObjectTable クラスのオブジェクトに格納される。ObjectTable クラスの add(Object) メソッドによりオブジェクトが追加されると、このメソッドは、オブジェクトに付加されたタグを返す。なお、このタグは、GHC 側から作成された個々のオブジェクトを識別するためのもので、GHC に渡して処理が出来るように、整数値で識別している。ObjectTable のオブジェクトには、新たなオブジェクトが生成されるたびに追加され、オブジェクト解放のメッセージによって、テーブルから除去される。多くのオブジェクトを効率的に扱うために、この ObjectTable は、ハッシュテーブルを用いてオブジェクトとタグの管理を行っている。

オブジェクトが生成され、テーブルに追加されると、オブジェクトのタグと、戻り値の設定先識別のタグとともに GHC 側にメッセージを送る。GHC 側では、input プロセスがこれを受け取る。そして、spool プロセスに、戻り値の設定先のタグとオブジェクトのタグを渡す。spool は戻り値のタグを元に、オブジェクトのタグを変数に設定する。そして、呼び出しの際に戻り値の設定を待っていたプロセスがオブジェクトに対応するプロセスを生成し、このプロセスへのストリームを、オブジェクト生成を行ったプロセスに渡す。これにより、各オブジェクトへのインタフェースが行えるようになる。

メソッドの呼び出しは、オブジェクト生成で新たに作成されたプロセスに対して、

method(メソッド名, 引数のリスト表現, 戻り値)

といったメッセージを送ることによりインタフェースを利用する。“メソッド名”は呼び出すメソッドのメソッド名を指定し、“引数のリスト表現”にメソッドの引数をリス

ト形式で指定する。“戻り値”には、メソッド呼び出しの戻り値を返すための変数を指定する。この変数には、正しくメソッドが呼び出された場合には、

normal(戻り値)

の形で戻り値を返す。また、メソッド呼び出しに失敗した場合には、

abnormal(エラーメッセージ)

の形式でエラーを渡す。

このメッセージをプロセスが受け取ると、まず、spool プロセスに、呼び出しの戻り値の設定先を登録し、これを識別するタグを受け取る。このタグと、オブジェクトのタグとともに、呼び出しに関するデータを Java 側に送信する。

Java 側では、ReadThread のオブジェクトがこのメッセージを受け取る。そして、オブジェクトの生成の場合と同様に、メソッド呼び出しのための新たなスレッドを生成し、ReadThread のオブジェクトは、他のインタフェースの処理を行う。

新たに生成されたスレッドは、まず、ObjectTable のオブジェクトから、指定されたタグを持つオブジェクトを取り出す。そして、このオブジェクトに対応する Class クラスのオブジェクトに対して、getMethod(String, Class[])や、getMethod()のメソッドを用いて、対応する java.lang.reflect.Method クラスのオブジェクトを取得する。なお、この際のメソッドを特定する際の引数の対応は、オブジェクト生成の場合と同様に、Data クラスに対してまず検索し、見つからなかった場合は String クラスや int 型を当てはめる。これにより、String クラスのオブジェクトや int 型のデータを引数に持つメソッドの呼び出しが簡単に行うことが出来る。

このようにして特定された Method クラスのオブジェクトに対して invoke(Object, Object[])メソッドを呼ぶことによりメソッドが実行される。そして、メソッド呼び出しの戻り値を、戻り値の設定先を示したタグとともに、GHC 側に送る。GHC 側では、input プロセスがこのメッセージを読み込み、このデータを spool プロセスに渡して、メソッド呼び出しの戻り値が設定される。

メソッドの戻り値は、基本的に Data クラスのオブジェクトを返すものを利用する。しかし、メソッドの引数の問題と同様に、これでは扱いにくい場合がある。そこで、引数の型変換と同様に、戻り値のクラスが String であった場合には、GHC の文字列型に、int 型の場合には、GHC の整数型に自動的に変換する。

フィールドの値の参照は、オブジェクト生成で作成されたプロセスに

get(フィールド名, フィールドの値)

といったメッセージを送ることによりインタフェースを利用する。“フィールド名”は参照するフィールド名を指定する。“フィールドの値”には、取得したフィールドの値を設定する変数を指定する。この変数には、フィールド値が正しく取得された場合には、

normal(フィールドの値)

の形で、フィールドの値を返す。また、フィールドの参照に失敗した場合には、

abnormal(エラーメッセージ)

の形式でエラーを渡す。

このメッセージを main プロセスが受け取ると、メソッド呼び出しの場合と同様の手続きを行い、Java 側にメッセージが送られる。Java 側では、ReadThread のオブジェクトがこのメッセージを受け取ると、新たなスレッドを生成し、フィールドの参照を行う。

このスレッドは、手続き呼び出しの場合と同様に ObjectTable のオブジェクトからオブジェクトの参照を取得する。そして、そのクラスのオブジェクトに対応する Class クラスのオブジェクトから、getField(String)を用いて java.lang.reflect.Field クラスのオブジェクトを取得する。

この Field クラスのオブジェクトの、get(Object)メソッドを用いて、フィールドの値を取得する。このとき、メソッド呼び出しの戻り値の扱いと同様に、フィールドの値が String クラスや int 型であった場合は、自動的に Data クラスのオブジェクトに変換されるようになっており、インタフェースの利便性を向上させている。取得したフィールドの値を GHC 側に返す処理は、メソッド呼び出しと同様の処理が行われる。

フィールドの値の設定は、オブジェクト生成で作成されたプロセスに

set(フィールド名, 設定値, 設定確認)

といったメッセージを送ることによりインタフェースを利用する。“フィールド名”は参照するフィールド名を指定し、“設定値”はフィールドに設定する値を指定する。“設定確認”には、フィールドの値が設定されたことを確認するための変数を指定する。この変数には、値が正しく設定されると、

normal

が設定される。また、フィールドの設定に失敗した場合には、

abnormal(エラーメッセージ)

の形式でエラーを渡す。この値を利用することにより、フィールドが正しく設定されたことを確認したり、フィールドが設定されたタイミングを知ることが出来る。

このメッセージを main プロセスが受け取ると、フィールド参照の場合と同様の処理を行い、Java 側にメッセージが送られる。Java 側では、ReadThread のオブジェクトがこのメッセージを受け取ると、フィールド参照の場合と同様に新たなスレッドを生成し、フィールド値の設定を行う。Field クラスのオブジェクトを取得すると、set(Object, Object)メソッドを呼び出して、指定されたデータを設定する。このとき、フィールド

のクラスが Data クラスでない場合は、メソッド呼び出し引数と同様の型変換が行われる。設定確認の値を GHC 側に返す処理も、メソッド呼び出しと同様の処理が行われる。

オブジェクトに対する操作が必要でなくなった場合は、Java で、そのオブジェクトに対する参照を消去すればよい。しかし、ObjectTable のオブジェクトが、個々のオブジェクトへの参照を保持しているため、テーブルからオブジェクトを取り除く操作を、GHC 側から明示的に行う必要がある。このタイミングは、オブジェクト操作へのインタフェースとなるプロセスへのストリームを閉じた場合となる。このとき、このプロセスは、オブジェクトをテーブルから取り除くように、Java 側にメッセージを送り、そのプロセスを終了させる。

Java 側でメッセージを受け取ったときに、新たにスレッドを生成し、そのスレッドがオブジェクト操作に関する処理をすると述べた。これは、他のインタフェースを用いた通信と並列に処理することが出来るようにしたためである。しかし、すべての処理を並列に行った場合に問題が生じる場合がある。例えば、あるメソッドを呼び出す指示を出した直後に、フィールドの値の参照を行うといったケースを考えてみる。このとき、メソッド呼び出しの中でフィールドの値が変更される場合であって、その処理に時間がかかる場合は、メソッドがフィールドの値を変更する前に、フィールドの値を参照してしまう場合がある。

このような問題は、同じオブジェクトに対して、同期を取っていないため生ずる。これを解決するには、メソッド呼び出しの戻り値を受け取るのを確認し、メソッドの終了を確認してから、次の処理を行うようにすればよい。しかし、いくつかのオブジェクト操作が重なる場合は、このような処理の記述が煩雑になる。また、GHC と Java の実行環境が離れていて、ソケットでのやり取りに時間がかかる場合は、インタフェース間での通信の時間に無駄が生じる。

そこで、同一のオブジェクトに関する操作に関して同期制御を行うこととした。これは、メソッド呼び出しやフィールドの参照を、インタフェースに指定された順序のまま、逐次的に処理するものである。これにより、オブジェクトに対するアクセスを、同時に GHC 側から送ることが出来るため、複数のメソッドを順番に処理する場合など、インタフェース間の通信の時間を無視することが出来る。また、オブジェクトごとに独立して同期処理を行うため、他のオブジェクトとは、並列に処理することが可能である。

オブジェクトの操作は、GHC 側から扱うために、Java のセキュリティ保護やスコープルールによって、使用できるクラスや、メソッド、フィールドが制限される。これは、インスタンスの作成や、メソッド呼び出しが、インタフェースの機能を提供する JK クラス等のクラスからアクセスされるため、これらのクラスから扱うことの出来ないクラスやメソッドは使用できないのである。つまり、オブジェクト操作で一般に利用できる

のは、publicに宣言されたクラスのオブジェクトの作成と、publicなメソッドや、フィールドへのアクセスである。

オブジェクト操作のインタフェースでは、クラスのインスタンスであるオブジェクトに対する操作だけでなく、クラスの持つ静的なメソッドや、静的なフィールドに対する開くアクセスも提供している。つまり、GHC側からクラスメソッド呼び出したり、クラスフィールドの参照、設定が行えるようになっている。

これらの操作を行う場合には、メッセージインタフェースのプロセスに対して、

class_method(クラス名, メソッド名, 引数のリスト表現, 戻り値)

class_field_get(クラス名, フィールド名, 戻り値)

class_field_set(クラス名, フィールド名, 設定値, 設定確認)

といったメッセージを送ることによりインタフェースを利用する。“クラス名”は利用するクラスのクラス名を指定する。残りの引数は、オブジェクトに対する操作と同様である。

これらのインタフェースの処理は、オブジェクトに対する処理と同様である。異なるのは、ObjectTableからオブジェクトを参照しないということと、他のクラスメソッド呼び出しなどの処理との同期を行っていない点である。オブジェクトメソッドの場合は、1つのオブジェクトに対する操作であったため同期処理が必要であったが、クラスメソッドは複数の処理を連続して行うする場合は少ないため、このような処理が必要ないと考えられるからである。

3.4 手続き呼び出しの実装

手続き呼び出しのインタフェースを利用して、GHC のゴールを呼び出す際の処理の流れについて述べる。手続き呼び出しのインタフェースを利用する場合には、JK クラスのオブジェクトに対して、

call(String module_name, String goal_name, Data args[]))

メソッドを呼び出す。ここで、“module_name”には呼び出すゴールのモジュール名を指定し、“goal_name”には呼び出すゴールのゴール名を指定する。また、“args”にはゴールに指定する引数のデータをDataクラスの配列で指定する。このメソッドは、手続き呼び出しの戻り値を返す。また、呼び出しが失敗した場合は、nullを返す。

このメソッドが呼ばれると、まず、メッセージインタフェースを作成したときに生成されたArgsManagerオブジェクトを利用し、この呼び出しの戻り値を識別するためのタグを取得する。そして、呼び出しデータに、このタグを付加して、GHC側に送信す

る。その後、ArgsManager オブジェクトを用いて、この手続き呼び出しの戻り値が得られるまで、メソッドの処理を一時停止させる。

GHC 側では、このメッセージを input プロセスが受け取る。そして、input プロセスは、このゴールを実行するためのプロセスを新たに生成し、メッセージの受け取りの処理に戻る。ゴールを実行するためのプロセスは、与えられたモジュール名とゴール名、そして、与えられた引数の個数に 1 を加えた引数を持つゴールを、GHC の実行環境である KLIC ジェネリックオブジェクトの機構を利用して取得する。得られたゴールに対して、Java 側から指定された引数に、戻り値として設定される変数を追加した引数の組を与え、リダクションを開始する。このゴールは、計算を開始して、その結果を戻り値として追加された変数に設定する。

ゴールを実行するプロセスは、新たに生成したゴールが、先に与えた戻り値に対応する変数に値が設定されるの待つ。値が設定されると、そのデータを、与えられた戻り値の識別タグとともに、Java 側に渡す。Java 側では、ReadThread のオブジェクトが、このメッセージを読み取り、ArgsManager にそのデータを渡す。ArgsManager は、戻り値の識別タグを元に、中断していたスレッドを特定し、そのスレッドに対して戻り値のデータを渡し、スレッドを再開させることにより、手続き呼び出しの処理が終了する。

3.5 通信プロトコル

GHC と Java 間のインタフェースが、ソケットを用いてメッセージをやり取りする際の内部プロトコル[6]について述べる。

インタフェースでやり取りされるメッセージの中身は、GHC のデータ型のデータである。これら GHC のデータを、ソケットでやり取りするために文字列に変換する必要がある。今回はこのための手法として、データを wrapped-term 形式で文字列に変換することとした。これは、データ型をファンクタ名とし、データの内容をその引数として付加して、ファンクタの形でデータを表現するものであり、

データ型名(データ内容)

というようになっている。GHC のデータのうち、アトム、整数、文字列は、このまま変換される。ファンクタ型のデータは、再帰的な構造になっているので、データの部分が再帰的に wrapped-term 形式に変換されたものとなり、

functor(ファンクタ名(第 1 引数の wrapped-term 形式表現, …))

といった形式になる。また、これらの変換の際に、改行文字、カッコ、引用符などは、バックスラッシュによる特殊記号のエスケープを行っている。具体的な例を図 7に示す。

データ	wrapped-term形式の表現
abc	atom(abc)
123	integer(123)
"qwerty"	string("qwerty")
f(10,"str")	functor(f(integer(10),string("str")))

図7 wrapped-term形式表現の例

それぞれのメッセージインタフェースの形態を用いた通信は、その機能の種類とデータをファンクタの形式で文字列に変換して送信する。

データストリームで、データを送る場合、

data(データ内容)

といった形式で文字列に変換されてソケットに書きこまれる。データ内容は wrapped-term 形式で文字列に変換される。

オブジェクト操作の場合、オブジェクト生成、メソッド呼び出し、フィールド値参照、フィールド値変更オブジェクトの解放のそれぞれで形式が異なる。オブジェクト生成の場合は、

instance(クラス名, 引数のリスト表現, タグ)

といった形式になる。引数はリスト形式のデータが渡される。タグは、オブジェクトを識別するためのタグを受け取るために付加されたタグである。メソッド呼び出しは、

method(オブジェクトのタグ, メソッド名, 引数のリスト表現, 戻り値のタグ)

といった形で変換される。戻り値のタグは、戻り値の設定先を識別するためのタグである。フィールド値の参照や変更は、

fget(オブジェクトのタグ, フィールド名, 戻り値のタグ)

fset(オブジェクトのタグ, フィールド名, 設定値, 戻り値のタグ)

のようになる。オブジェクトの解放は、

remove(オブジェクトのタグ)

という形式になる。

クラスメソッドの呼び出しや、クラスフィールドの参照、変更は、

class_method(クラス名, メソッド名, 引数のリスト表現, 戻り値のタグ)

class_fget(クラス名, フィールド名, 戻り値のタグ)

class_fset(クラス名, フィールド名, 設定値, 戻り値のタグ)

のようになっている。

手続き呼び出しのインタフェースで、GHC のゴールを生成する場合は、

call(モジュール名, ゴール名, 引数のリスト表現, 戻り値のタグ)

のようになる。

具体的な呼び出しの例を図 8に示す。

内容	送信される文字列
データストリームで f(12) というデータを送る	`data(functor(f(integer(12))))`
Sleep クラスのオブジェクトを0引数のコンストラクタで生成する 戻り値のためのタグは1	`instance(string("Sleep"),atom([]),integer(1))`
hanoi モジュールの solve というゴールを1引数 "4" を用いて生成する 戻り値のためのタグは2	`call(string("hanoi"),string("solve"),functor(.(integer(4),atom([]))),integer(2))`

図8ソケットで送られるメッセージの例

第4章　インタフェースの利用例

4.1 データI/O

データストリームのインタフェースを利用した例としてデータ I/O を作成した。

これは、GHC のプログラムに、Java による GUI を用いてデータを与えるプログラムとなっている。GUI では、データの入力や表示を図形的表現で行うことが出来るようになっている。

GHC 側のプログラムは、サーバプログラムとして記述してあり、その主な部分は、図 9に示すようになっている。

```
mainProcess(Socket) :-
        jk:jk(JKS, Sock),
        echo(JKS).

echo(S) :-
        S = [get(Data)|Rest],
        echoSub(Rest, Data).

echoSub(S, D) :- D = normal(Data) |
        S = [put(Data) | ST],
        echo(ST).

echoSub(S, D) :- D = abnormal(_) |
        S = [close].
```

図9 GHC 側のプログラムの主な部分（データ I/O）

mainProcess の引数に与えられたソケットのストリームを利用し、jk:jk/2 を用いてインタフェースを作成している。echo/1 により、まずデータストリームからデータを 1 つ取得し、echoSub/2 に分岐する。ここで、データが取得できた場合は、1 つ目の節により、データストリームのインタフェースを用いて、取得したデータが Java 側に渡される。そして、再び echo/1 を呼び出す。データの取得に失敗した場合は、2 つ目の節により、インタフェースが終了し、このプロセスが終了する。

Java 側のプログラムは、Java AWT を用いた GUI を持つプログラムとなっており、クライアントプログラムとして動作する。実行画面は図 10のようになる。

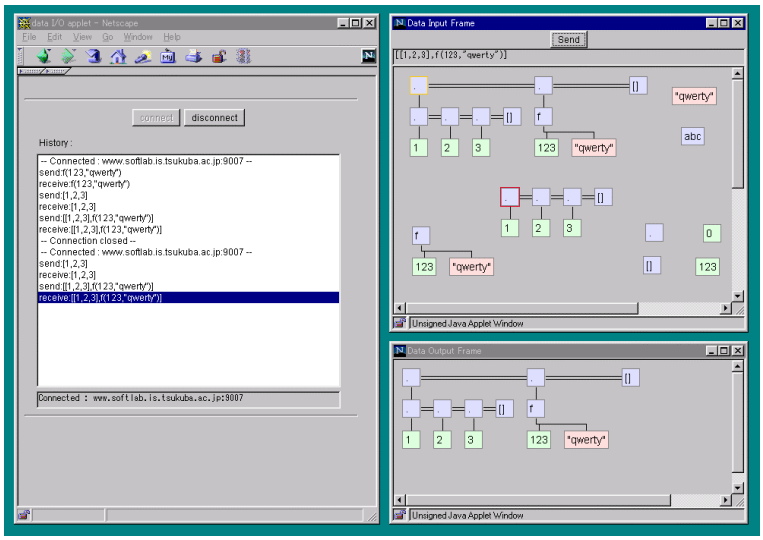

図10 Java 側のプログラム実行画面 (データ I/O)

GUI は、メインウィンドウ、データ入力ウィンドウ、データ出力ウィンドウからなる。メインウィンドウ（図 11）には、GHC のプログラムとの接続を行うための”connect”ボタンと”disconnect”ボタンがあり、中央部には、データストリームでやり取りされたデータが表示されているヒストリがある。

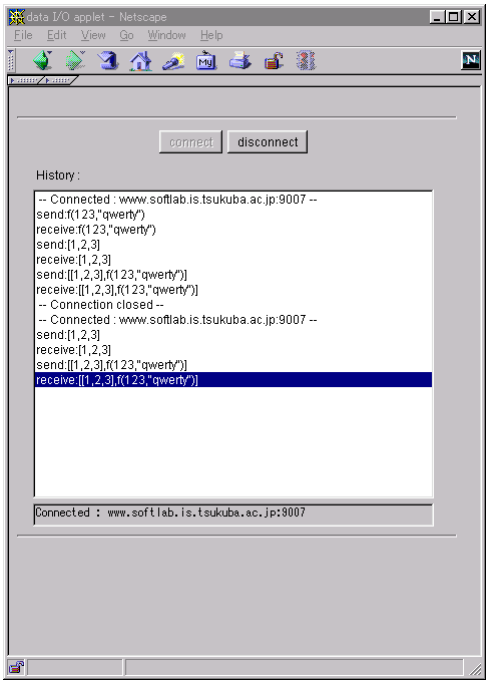

図11 メインウィンドウ

データ入力ウィンドウ（図 12）では、GHC 側のプログラムに送るデータを、図的表現を利用して入力する。中央にあるキャンバスで、マウス操作によりデータを入力する。また、上部の”Send”ボタンを押すと現在選択されているデータが GHC 側に送られる。

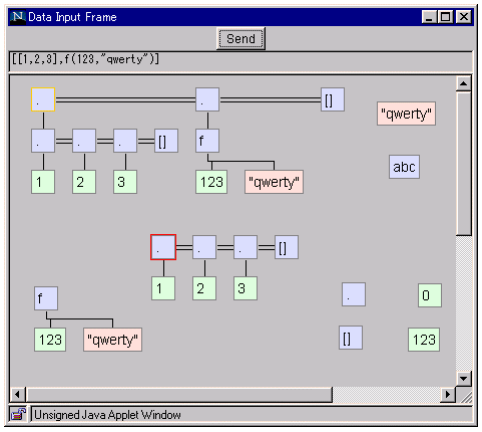

図12 データ入力ウィンドウ

データ出力ウィンドウ(図 13)では、GHC から送られたデータを図的表現で表示している。また、メインウィンドウのヒストリを選択することによって、これまでの送受されたデータを見ることも出来る。

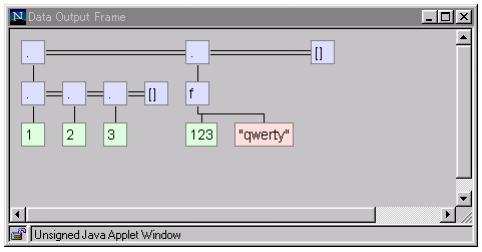

図13 データ出力ウィンドウ

メインウィンドウの”connect”ボタンが押されると、ソケットを接続し、JK クラスのコンストラクタを用いてインタフェースが作成される。また、GHC 側から送られてくるデータを常に監視するためのスレッドを作成する。

このスレッドのプログラムを、図 14の示す。このスレッドは、データストリームからデータを読み込み、そのデータを出力ウィンドウに追加し、ヒストリにそのデータを登録している。図 14中の jks は、インタフェースを提供する JK クラスのオブジェクトである。kdo はデータ出力ウィンドウのオブジェクトで、setData(Data)メソッドにより、受け取ったデータが表示される。history_data, history は通信履歴を管理するオブジェクトである。closeJKS()メソッドは、インタフェースの終了作業を行うメソッドであり、jks オブジェクトに対して、close()メソッドを呼び、GUI の設定を行う。

```
class GetThread extends Thread {
  public void run() {
    for(;;) {
     if (jks == null)
       return;

     Data d = jks.get();
     if (d == null) {
       closeJKS();
       return;
     }

     history_data.addElement(d);
     String str = "receive:" + Data.toStr(d);
     history.addItem(str);
     kdo.setData(d);
    }
  }
}
```

図14 Java 側のデータ読み込みスレッド

データ入力ウィンドウで”Send”ボタンが押されると、アクションに関連付けられていたメソッドが呼ばれる。このメソッドは、図 15のようになっている。まず、データ入力ウィンドウから選択されたデータを取得する。そのデータをデータストリームで GHC 側に送信し、そのデータをヒストリに登録している。図 15中の kdi はデータ入力ウィンドウのオブジェクトで、getSelectData()メソッドにより選択されたデータを取得している。

```
public void actionPerformed(ActionEvent event) {
 if (event.getSource() == send) {

  Data d = kdi.getSelectData();
  if (d != null && jks != null) {
    jks.put(d);

    history_data.addElement(copy_data);
    String str = "send:" + Data.toStr(d);
    history.addItem(str);
  }
}
```

図15 Java 側のデータ送信メソッド

GHC 側のサーバプログラムを実行した後、Java のプログラムを起動する。GHC のプログラムと接続して、データ入力ウィンドウからデータを設定し送信すると、GHC 側のプログラムは、そのままデータをデータストリームのインタフェースで送り返す。Java 側ではそのデータが受け取られて、データ出力ウィンドウに表示される。このよ

うにして、データストリームのインタフェースを用いたアプリケーションが作成できる。

なお、この例で用いている Java のプログラムは、データストリームを用いた通信のみ行うため、GHC 側のプログラムは、どのようなプログラムにも利用できる。GHC 側で、データストリームのインタフェースを用いてデータをやりとるするようにプログラムをしておけば、この GHC のプログラムに対して、インタラクティブにデータを与えることが可能となる。また、図的表現を用いてデータの入出力を行うため、非常に簡単に GHC のデータを扱うことが可能となっている。

この例で用いている Java のプログラムは、単独のアプリケーションとして使用するだけでなく、アプレットとして HTML の中に貼りつけることも可能である。このことにより、WWW を介して GHC のアプリケーションが利用できる、グラフィカルインタフェースを提供することが出来る。

4.2 インスペクタ

オブジェクト操作のインタフェースを利用した例として、インスペクタを作成した。

これは、GHC プログラムの内部動作の情報を、Java の GUI に出力させるものである。GHC 側では、GUI を生成する Java のオブジェクトを、オブジェクト操作のインタフェースで作成し、このオブジェクトのメソッドを用いて、GHC のプログラムの内部動作を GUI に表示、確認することが出来る。

インスペクタの GHC 側のプログラムは、ある GHC プログラムの動作情報を、ストリームを使って受け取り、そのデータをオブジェクト操作のインタフェースを用いて Java の GUI に出力する。このプログラムは図 16のようになっている。

```
inspector(S, JK) :- S = [] |
      JK = [].

inspector(S, JK) :- S = [new(Name, Stream) | ST] |
      JK = [instance("Inspector", [Name], Ret) | JKT],
      inspectorSub(Stream, Ret),
      inspector(ST, JKT).

inspectorSub(Stream, Ret) :- Ret = normal(ObjectStream) |
      inspectorMain(Stream, ObjectStream).

inspectorSub(Stream, Ret) :- Ret = abnormal(Message) |
      true.

inspectorMain(S, OS) :- S = [] |
      OS = [].

inspectorMain(S, OS) :- S = [put(Data) | ST] |
      OS = [method("insertData", [Data], Ret)|OST],
      inspectorMain(ST, OST).

inspectorMain(S, OS) :- S = [write(Data) | ST] |
      OS = [method("insertString", [Data], Ret)|OST],
      inspectorMain(ST, OST).

inspectorMain(S, OS) :- S = [nl | ST] |
      OS = [method("insertString", ["¥n"], Ret)|OST],
      inspectorMain(ST, OST).
```

図16 GHC 側のプログラムの主な部分（インスペクタ）

ここで、inspector/2 の受け取る引数は、S にプログラムの内部状態を受け取るストリームをリストで指定し、JK には Java のプログラムと接続するためのメッセージインタフェースへのストリームを指定する。inspector/2 に指定したストリームに対して、put(*Data*), write(*String*), nl 等のメッセージを送ると、Java の GUI に指定されたデータが出力され、GHC プログラムの内部状態を表示させることが出来る。

inspector/2 の 2 番目の節で、instance/3 のメッセージにより Inspector クラスのオブジェクトを生成している。そして、inspectorMain/2 の各節により、method/3 のメッセージを用いて、2 種類のメソッドの呼び出しを行っている。

Inspector クラスは、GHC の内部動作を表示する GUI を作成する。実行画面は図 17 のようになる。GHC からは、

```
public String insertData(Data)

public String insertString(String)
```

のメソッドが利用される。insertData(Data)メソッドは、引数に与えられたデータを文字列表現で表示し、表示した文字列を返すものである。insertString(String)は、引数の文字列を表示し、その文字列を返すメソッドである。

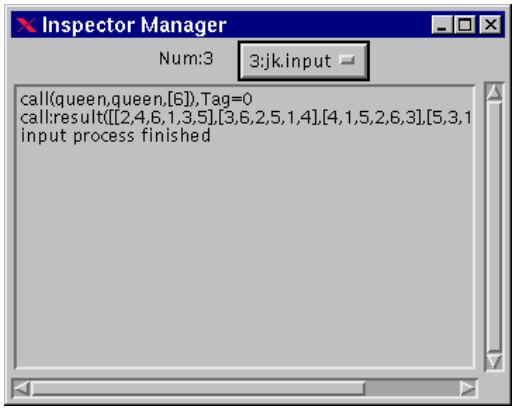

図17 Inspector クラスの GUI

4.3 ハノイの塔

手続き呼び出しのインタフェースを用いた例としてハノイの塔のプログラムを作成した。

このプログラムは、ハノイの塔の GUI を Java で記述してあり、ハノイの塔を解く手順を求めるプログラムを GHC で記述してある。GHC の解を求めるプログラムを、Java 側から手続き呼び出しのインタフェースを用いて利用している。

GHC 側のプログラムは、サーバプログラムとして動作し、ソケットが接続されると、そのソケットを用いてメッセージインタフェースを作成する。このプログラムの主な部分を図 18に示す。このプログラムでは、jk:jk/2 でインタフェースを作成しているが、JKS=[]となっており、インタフェースに対してメッセージを送っていない。つまり、GHC 側からはこのインタフェースに対してなにも操作を行わない。

```
mainLoop(ServerSocket) :-
      ServerSocket = [accept(AcSock) | SST],
      mainLoop(SST, AcSock).

mainLoop(ServerSocket, AcSock) :-
            AcSock = normal(Sock) |
      jk:jk(JKS, Sock),
      JKS = [],

      mainLoop(ServerSocket).
```

図18 GHC 側のプログラムの主な部分(ハノイの塔)

また、メインプログラムとは別に、ハノイの塔を解くプログラムが記述してあり、手続き呼び出しのインタフェースを用いて、Java 側から利用される。これは、hanoi モジュールの hanoi ゴールと、そのサブプログラムである。Java 側から利用されるゴールは、

hanoi:hanoi *+Data +To -Ans*

である。Data に現在の円盤の状況を指定し、To に移動したい棒の番号を指定してリダクションすると、Ans に円盤の動かし方が設定される。例えば、Data が[[3,2,1],[],[]]で、To が 1 であったとすると、Ans には [move(0,1), move(0,2), move(1,2), move(0,1), move(2,0), move(2,1), move(0,1)] が設定される。

Java 側のプログラムは、Java AWT を用いた GUI を持つプログラムとなっており、クライアントプログラムとして動作する。実行画面は図 19のようになる。

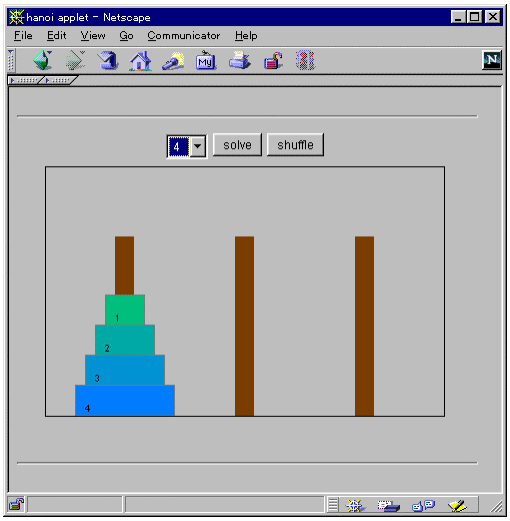

図19 Java 側のプログラムの実行画面(ハノイの塔)

上部には、円盤の枚数を表示している部分、”solve”ボタン、”shuffle”ボタンがある。”solve”ボタンを押すと、GHC のプログラムと接続し、ハノイの塔を解く手順を取得する。”shuffle”ボタンを押すと、円盤の並び方がでたらめに変更される。

“solve”ボタンを押すと、アクションに関連付けられたメソッドが呼ばれる。このメソッドの主な部分は、図 20のようになっている。ここで、図 20中の h_canvas はハノイの塔のキャンバスであり、getMaxSizeDiskPole()メソッドで、一番大きい円盤のある棒の番号を取得している。encodeData()メソッドは、現在の円盤の位置情報取得し、Data クラスのオブジェクトとして返すメソッドである。makeJKS()メソッドは、ソケットを接続し、JK クラスのオブジェクトを返すメソッドである。Animation クラスは、コンストラクタの引数に渡された、ハノイの塔を解く手順に従い、キャンバス上の円盤を移動させるアニメーション表示を行うスレッドである。

```
int to = (h_canvas.getMaxSizeDiskPole() + 1) % 3;

Data disk_data = encodeData();
Data to_data = Data.Integer(to);

JK jks = makeJK();
if (jks == null) {
  return;
}

Data args[] = new Data[] {disk_data, to_data};
Data ret = jks.call("hanoi", "hanoi", args);

jks.close();

new Animation(ret).start();
```

図20 解を取得するメソッドの一部

call(String, String, Data[])メソッドにより、手続き呼び出しのインタフェースを用いて、GHC と接続し、hanoi:hanoi/3 ゴールを呼び出している。そして、メソッドの戻り値として、ハノイの塔を解く手順を取得し、この Data クラスのオブジェクトを ret に設定している。

なお、この例で用いている Java のプログラムは、単独で利用する他に、アプレットとしても利用できるようになっている。ハノイの塔を解くプログラムを含んだ GHC のサーバプログラムを、WWW サーバの動作しているホストで実行しておき、この Java のアプレットを含んだ HTML ファイルを開くことにより、ハノイの塔のプログラムが利用できる。ユーザが、”solve”ボタンを押すと、メッセージインタフェースを作成し、GHC のプログラムと接続して解を取得することが出来る。

第5章　他の言語間インタフェース

5.1 JIPL

JIPL[7]は、(株)アイザックの紀信邦によって作成された Java と Prolog とのインタフェースである。

JIPL は、Java Interface for Prolog の略で、Java アプリケーションから Prolog の述語を呼び出したり、Prolog 言語で記述されたプログラムから Java 言語のメソッドを呼び出し、フィールドにアクセスすることが出来る。

JIPL で用いられる Prolog は、アイザックの製品である K-Prolog を実行環境とした Prolog である。JIPL は、K-Prolog と接続するために、Java の JNI（Java ネイティブインタフェース）を利用している。JNI は、Java の提供する機能で、Java のプログラム中で、C 言語等で記述されたプログラムを利用するためのインタフェースである。JIPL は、JNI を利用して、K-Prolog の実行環境と Java インタプリタを接続し、Prolog と Java のプログラムを結びつける。

JIPL を用いたプログラムは、1 つのアプリケーションとして動作する。Java 側の実行プログラムを起動すると、JIPL が JNI を利用して K-Prolog の実行環境を作成する。Prolog プログラムを利用する場合は、JIPL が K-Prolog を用いて Prolog のプログラムを実行する。

JIPL を用いて、Prolog のプログラムを Java のプログラムから利用す場合には、isac.plc.Plc クラスを利用する。まず、このクラスの静的メソッドである、

```
public static void startPlc(String args[])
```

を用いて、K-Prolog を起動する。この後は、

```
public static native boolean exec(String command)
```

を利用して K-Prolog の組み込み述語を実行することが出来る。また、PLC クラスのインスタンスを作成することにより、実行したいゴールを得ることが出来る。このオブジェクトに対して、

```
public boolean call()
```

メソッドを呼び出すことにより、Java から Prolog のプログラムが実行できる。

JIPL を用いて、Prolog のプログラムから Java のプログラムを利用する場合には、JIPL の提供する述語を利用する。これらは、

```
javaConstructor(Class(Args,...), Instance)

javaMethod(ClassOrInstance, Method(Args,...), Return)

javaGetField(ClassOrInstance, Field, Value)
```

```
javaSetField(ClassOrInstance, Field, Value)
```

といった形式で利用される。これらの述語を用いることにより、オブジェクトを作成し、そのオブジェクトのメソッドを呼び出したり、フィールドの値を参照することが出来る。また、クラスメソッドや、クラスフィールドを利用することも可能である。

JIPL は、JNI を用いて Java と Prolog を接続している。よって、JIPL 内部では、関数呼び出しの形で通信が行われている。また、Java から Prolog のプログラムに対して操作が行えるのは、述語の呼び出しという形に限られている。

5.2 MINERVA

MINERVA[8]は、IF Computer 社が作成した、Java 上で動作する Prolog の実行環境を提供する製品である。

Prolog のソースプログラムをコンパイルし、実行させることが出来る。MINERVA は、Prolog のプログラムを独自のオブジェクト形式に変換する。このオブジェクトコードを実行するのは、Java のプログラムとなっており、Minerva クラスとして実装されている。このクラスは、先のオブジェクトコードを読み込んで実行する。Minerva クラスは、コマンドラインからの利用の他に、アプレットからも利用することができる。

MINERVA のシステムは、Prolog と Java の言語間のインタフェースというよりは、Prolog プログラムから、Java へのトランスレータであるといえる。このトランスレータが MINERVA コンパイラである。

MINERVA では、Prolog のプログラム中で、Java のプログラムが利用可能となっている。Java のプログラムを利用するには、GUI などの既に提供されている機能を利用する場合と、そうでないクラスを扱うための機構が存在する。前者の場合、例えば、

```
frame__create / 1, frame__dispose / 1, frame__pack / 1,

checkbox__create / 2, checkbox__create / 4, checkbox__get_label / 2,

checkbox__get_state / 1, checkbox__set_label / 2, checkbox__set_state / 2,
```

等の述語を用いることにより、Prolog のプログラム内で、Java のプログラムが利用できる。

また、このように既に用意されているものでない場合は、Java アクセスインタフェースという機能を用いて、新たな述語を生成して利用する。Prolog のプログラム中で利用したいメソッドに関する情報を、別のファイルを用いて定義する。これにより、必要な述語が自動的に生成される。

MINERVA の Prolog プログラムは、Java のプログラム中から利用することが出来る。これは、先に挙げた Minerva クラスのオブジェクトを利用する。このクラスのオブジェクトに対して、

```
public void execute(String functor) throws MinervaSystemError
```

public void generic_execute(String functor, Object[] args) throws MinervaSystemError

というメソッドを用いることで、指定された述語が実行される。このようにすることで、Java から、Prolog のプログラムが利用できる。

MINERVA は、Prolog のプログラムを Java で利用できる形にトランスレートすることにより、Prolog のプログラムを Java の実行環境で利用し、Java プログラムと接続することが可能となるっている。Java と Prolog のプログラム間の通信は、Java オブジェクト間の通信に変換されることになる。また、Prolog のプログラムは Java に変換されるため、1 つのアプリケーションとして動作させることが出来る。

5.3 klitcl

klitcl[9]は、南雲淳によって作成された GHC と Tcl/Tk のそれぞれのプログラムを接続するインタフェースである。

GHC の実行環境である KLIC と Tcl/Tk を起動するサーバプログラムを接続して通信を行う。GHC 側は、このインタフェースを用いたクライアントプログラムを作成する。Tcl/Tk 側は、klitcld というサーバプログラムが動作している。このプログラムは C 言語で記述されており、GHC のプログラムが接続すると、Tcl/Tk のインタプリタを起動し、通信を行う。

GHC 側から Tcl/Tk のプログラムを利用する場合は、インタフェースを初期化した後に、

klitcl:tcl_eval *+Command –Result +InStream –OutStream*

klitcl:tcl_evalfile *+FileName –Result +InStream -OutStrem*

klitlc:tk_mainloop *+InStream -OutStream*

klitcl:tk_wait *+InStream -OutStream*

といったゴールを用いて通信を行う。これらのゴールを呼び出すことにより、Tcl/Tk のコマンドを実行したり、Tk のイベントを待つことが出来る。

また、Tcl/Tk のプログラム中で、GHC 側のプログラムを利用する際には、klitcl の提供する Tcl コマンド、

klic *module_name predicate_name ?arg ...?*

を利用する。このコマンドは、GHC 側で tk_mainloop か tk_wait のいずれかのゴールが呼ばれている場合に、コマンドの引数に指定されたゴールを実行するものである。

klitcl は、GHC 側のメインプログラムと、Tcl/Tk を実行するサーバプログラムの 2 つのプログラムから構成されている。これらは、Unix ドメインソケットで接続される。ソケットの制限により、この 2 つのプログラムは、同一ホスト上でなければ動作することが出来ない。

Tcl/Tk 側から、GHC のプログラムを利用するには、GHC のゴールを呼び出すというインタフェースを用いる。このとき GHC に渡すことが出来るデータは、文字列データに制限されている。また、GHC 側が tk_wait か tk_mainloop のゴールを呼び出していなければ、Tcl/Tk 側からアクセスすることは出来ない。

GHC 側から Tcl/Tk のプログラムを利用する場合は、tcl_eval か tcl_evalfile のゴールを呼び出す。これらは、tk_wait や tk_mainloop による待機状態では利用することが出来ず、インタラクティブなアプリケーションを作成するのは難しい。

5.4 その他

KLIJava[10]は、GHC のプログラムを Java のプログラムに変換するトランスレータである。変換されたプログラムは単独でのみ利用でき、GHC のプログラムから Java のプログラムを利用したり、Java のプログラムから、GHC のプログラムを利用することは出来ない。

Prolog のプログラムを Java で実行できるようにしたものは、Prolog Cafe[11], jProlog[12], W-Prolog[13]などがある。

Prolog Cafe は、Prolog のプルグラムを Java のソースプログラムに変換することにより Java の実行環境で Prolog のプログラムが利用できるようにしたものである。変換されたプログラムは単独でのみ利用でき、Prolog のプログラムから Java のプログラムを利用したり、この逆に、Java から Prolog のプログラムを利用するようなインタフェースは持っていない。

jProlog は、Java 記述された Prolog の実行環境であり、CUI で Prolog が実行できる。W-Prolog も、Java で記述された Prolog の実行環境であるが、ユーザインタフェースに GUI を用いている。この実行環境は、単独のアプリケーションとして利用する他に、アプレットとして、HTML に貼りつけて利用することも可能である。jProlog と W-Prolog は、Prolog のプログラムが実行できるだけで、Java のプログラムとのインタフェースは持っていない。

第6章　まとめ

GHC と Java の 2 つの言語で記述されたプログラム間で通信を行うためのメッセージインタフェースについて検討した。このインタフェース形態として、データストリーム、オブジェクト操作、手続き呼び出しの３つのインタフェースを提案した。これらのインタフェースは、各言語で利用しやすいインタフェース形態を持ち、また、それぞれの言語の機能を使うための有効なインタフェースとなる。

また、このインタフェースの実装について触れ、全体構成と、そのインタフェース内部処理について述べた。また、インタフェースを利用した例を挙げ、インタフェースがどのように利用されるかを示した。

このインタフェースを利用することにより、GHC と Java とい 2 つの言語を利用したアプリケーションの構築が容易になり、GHC アプリケーションの GUI を Java で提供するといったシステム等、様々な分野のアプリケーションに適用で可能である。

参考文献

[1] Kazunori Ueda, "Guarded Horn Clauses," ICOT Technical Report TR-103, ICOT, 1985

[2] AITEC, “KLIC’s page” http://www.icot.or.jp/AITEC/COLUMN/KLIC/klic-J.html

[3] Javasoft, “Java Technology Home Page” http://www.javasoft.com/

[4] Ehud Shapiro, Akikazu Takeuchi, "Object Oriented Programming in Concurrent Prolog," New Generation Computing, Vol.1, No.1, 1983, pp.25-48

[5] Andrew D. Birrell, Bruce Jay Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer System, Vol.2, No.1, February 1984, pp39-59

[6] Alfred Z. Spector, “Performing Remote Operations Efficiently on a Local Computer Network”, Communications of the ACM, Vol.25, No.4, April 1982, pp.246-260

[7] Isac, "Java Interface for Prolog", http://prolog.isac.co.jp/jipl/index.html

[8] IF Computer, "MINERVA" http://www.ifcomputer.com/MINERVA/

[9] AITEC, "H8 itaku-kenkyu catalogue-02", http://www.icot.or.jp/AITEC/FGCS/funding/96/catalogue02.html

[10] Satoshi Kuramochi, “KLIJava Home Page” http://www.ueda.info.waseda.ac.jp/~satoshi/klijava/

[11] Mutsunori Banbara, "Java Prolog” http://pascal.seg.kobe-u.ac.jp/~banbara/PrologCafe/index.html

[12] Bart Demoen, "PrologInJava" http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/

[13] Michael Winikoff, “W-Prolog” http://www.csee.wvu.edu/~winikoff/wp/