LIR : COINS プロジェクトの低水準中間言語

阿部正佳[†] 藤波順久[††]
中田育男[†††] 萩谷昌己[†]

本発表では COINS プロジェクトの低水準中間言語 LIR の特徴的な部分を説明する．中間言語 LIR の目的は GCC の RTL と同様であり、また設計も似ている．一方で LIR は RTL にはない以下のような特徴を持つ．

まず関数、ブロック、モジュールという、通常の低水準中間言語には見られない高水準の構造を導入することにより、最適化からコード生成に至るパスをあるモデルの上でのプログラム変換として捉えることを可能とした．また LIR のドキュメンテーションでは表示的意味論で LIR の意味を記述した．これにより先のモデルが厳密に定義され、またポータブルで信頼性の高い処理系が実現可能となる．LIR は RTL 同様揮発性オブジェクトの表現を持つが、我々はその意味も厳密に定義しているので、揮発性オブジェクトへの誤った最適化を禁止出来る．実行中のプログラムは揮発性変数をチャネルとしたプロセスと見なせる．この考えにより揮発性オブジェクトのもうひとつの、そしてより自然な意味付けが出来るが、一方でそれは大雑把であり実用的ではない．しかし、揮発性オブジェクトに対するこれら二つの意味は等価であることが示せるので、我々の意味定義は妥当なものと言える．

LIR : A Low-level Intermediate Language in COINS project

Seika Abe ,[†] Nobuhisa Fujinami ,[††] Ikuo Nakata [†††]
and Masami Hagiya[†]

In this presentation, we briefly introduce essential advantages of the low-level compiler intermediate language LIR used in the project. The purpose and basic concepts of LIR are similar to those of RTL which is the intermediate language of GCC. The advantages of LIR are as follows.

First, by employing higher-level constructs, usually absent in low-level compiler intermediate languages, we can describe all of compiler back-end passes, from code optimizations to code generations, as program transformations on a model. Secondly, the documentation of LIR is written by a formal method based on denotational semantics. This will ensure the existence of the model and help implementers write portable and reliable compilers. Finally, LIR has a representation for volatile objects like RTL, but also we rigorously define a semantics for the objects. This will inhibit wrong optimizations for the objects. A running program can be regarded as a process communicating via channels of volatile variables. This view inspires another and more natural semantics of volatile objects; this semantics is, however, too rough to be practically useful. And a proof of equivalence between the two semantics provides adequacy of our semantics for the objects.

1. 導　　入

COINS プロジェクト☆は、各種のコンパイラの研究や開発のための共通インフラストラクチャを開発するプロジェクトであり、文部科学省科学技術振興調整費の補助を受けている．

LIR は COINS プロジェクトの低水準中間表現の名称であり、Low-level Intermediate Representation の略である．以後中間表現ではなく、中間言語という

† 東京大学情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo.
†† ソニー株式会社ネットワーク＆ソフトウェアテクノロジーセンターアドバンストテクノロジー開発部
Advanced Technology Development Department, Network & Software Technology Center, Sony Corporation
††† 法政大学情報科学部
Faculty of Computer and Information Sciences, Hosei University

☆ http://www.coins-project.org/

言葉を使う．LIR はいわゆるコンパイラバックエンドと呼ばれる部分で共通に使われる中間言語であり、コード最適化から最終的な機械語生成までの各フェーズにおいて使われる．

我々は多言語 (マルチソース)、多機種 (リターゲッタブル) コンパイラを目指している．多言語サポートに関しては HIR (High-level Intermediate Representation) という中間言語があり、少なくとも C、Fortran、Java の三つのソース言語を表現することを目指している．HIR については本論文では言及しない．

中間言語を、基本的な演算ノードからなるツリーで表現し、それを各マシン命令を表すツリーでパターンマッチし、マシン固有の命令に分解することによりコード生成を行なうという構成は既にリターゲッタブルコンパイラの確立された技術であり[2)8)9)4)]、我々もこのアプローチに従う．

GCC (GNU C Compiler)[5)] もこのアプローチに基づくリターゲッタブルコンパイラであり、その中間言語は RTL (Regiser Transfer Language) と呼ばれる．最近では厳密に GCC の RTL ではないのに、同様の表現を RTL と呼ぶこともあるようで、GCC の影響力を感じる．GCC の成功の大きな理由の一つは明らかにこの RTL の簡潔で明解なデザインにある．

多くのコンパイラの中間言語はインプリメンタのみ知り得るものであり、よって、その意味はアドホックに多義的なものになりがちである．その点、GCC の RTL はその名前どおり正式な言語として設計されている．それは S 式によるシンタックスを持ち、またフォーマルではないが十分に厳密な意味定義もされており、しっかりしたドキュメントも存在する．

LIR の目的は RTL と同じといってよい．また LIR は S 式によるシンタックス定義、メモリー参照の考え方など RTL と同様の部分も多い．以後 RTL については既知であると仮定し、LIR において特徴的な部分についてのみ解説を行なうのが本論文の目的である．

$$\text{LIR} = \text{RTL} + \text{上位構造} + \text{表示的意味定義}$$

ここで上位構造とはモジュールと多値関数インタフェースからなる．GCC の RTL は実行可能コードのみの表現であった．しかし中間言語を正式なプログラミング言語として設計するにはデータの表現も含めた一つのコンパイル単位（モジュールと呼ぶ）に対する表現が必要になってくる．多値関数については後に説明する．

また、表示的意味定義[18)16)] とは LIR のセマンティックスがフォーマルに定義されていることを意味する．どんな言語であれ、言語仕様は厳密であるに越したことはないが、リターゲッタブル性を考慮した中間言語において厳密な定義は本質的である．とくに COINS ではつぎの理由も上げられる．

COINS のインプリメント言語 Java[7)] は命令の仕様が厳密に決められており、どのマシンで動く Java も同じ命令は同じ意味を持つ．したがってあるターゲットマシン上で、そのマシンのオブジェクトコードを生成する、という状況においてすら、COINS コンパイラは本質的にクロスコンパイラである．すなわち Java のある演算は、一見対応するように見えるターゲットマシンでの演算と振舞いが異なる可能性があり、典型的には定数畳み込みなどでバグの原因となりかねない．実際クロス環境で動作する GCC のバグは基本的に RTL の意味定義の曖昧さが起因していると考えられる．このような危険性を回避するために命令の意味を厳密に定義する必要が生じる．

しかしこの問題は単に何らかの厳密な定義を与えればよい、という問題ではない．すなわち、リターゲッタブルコンパイラである以上、Java のようにターゲットマシンを、こちらで勝手に定義することは出来ない．アドレッシングモードの多様性は、たしかに基本演算の組合せで吸収出来るが、ここで問題なのは加算とか乗算といった基本演算である．例えば 2 の補数表現の計算機と 1 の補数表現の計算機では同じ整数加算でも異なる意味となる．

単に表現力のみ追求するならば、基本演算を増やすなり多義にすればよいが、そのために命令の簡潔さが失われ、インプリメントは複雑になる．我々は現在主に使われているハードを考慮して、メモリがバイトアドレスであり、整数の表現は 2 の補数であるという仮定を置くことにした．すなわち、そうでないハードは COINS ではサポートしない．

そのような仮定を置いても尚、意味をこちらでひとつに定義出来ない命令が残った．それは Java と C に存在するシフト命令である．そのために LIR のシフト命令は現行のマシンにおける食い違いをカバー出来るような表現となっている．

C 言語にある volatile 変数の考えは重要なものであり、LIR のメモリアクセス式は volatile オブジェクトを表現出来る．同様の表現は RTL にも存在するが、我々はその表示的意味を定義し、その定義が自然であることを示す．

関数呼び出しはリターゲッタブルコンパイラで問題になる部分のひとつである．ターゲットマシンを固定したとしても尚、さまざまなコーリングシーケンスがあり得る．引数の一部は通常レジスタに割り当てられ

るが、それは後にレジスタアロケータが自由に割り当ててよいものではなく、関数呼び出しの規約に従った割り当てをしなければならない．GCC では RTL 生成時にコーリングシーケンスを展開してしまう、という方法を取っている．後に議論するように、これは一つの極端な方法であり、最適化のクオリティーは期待できるが、コード最適化が複雑になる． 一方我々は LIR の関数呼び出しを高級言語のようにパラメータを明示する表現としており、RTL とのアプローチと逆のメリット、デメリットを持つ．

1.1 論文の構成

LIR の言語仕様はまとめて付録とし、本文では LIR の特徴的な部分の解説を行なう．セクション 2 では基本演算と値の表現について説明する．セクション 3 ではシフト命令に関する議論を行なう．セクション 4 では volatile オブジェクトの意味定義とその妥当性を論じる．セクション 5 では多値関数の導入と割り当て済みレジスタについて論じる．セクション 6 では例により LIR を概説する．セクション 7 はまとめである．GCC の RTL についての予備知識がない場合は、まずセクション 6 の例を最初にざっと眺められることを勧める．言語仕様で使われる表記を本文中でも使用する．標準的でないものについては、付録の「表記法のまとめ」を参照されたい．

付録の「LIR の仕様」では本質的でないような細かい定義や説明を一部省略している．LIR の正式なドキュメントは COINS のホームページにある．

2. 基本演算

厳密な意味論が与えられている言語としては、高級言語において Scheme[6] や ML[15]、またコンパイラの中間言語において SML[13] がある．しかしいずれも加算や乗算といった基本命令の仕様を厳密には規定していない．コンパイラの最適化変換は各演算が満たす代数的関係に基づいて行なわれるため、それら演算の仕様が厳密に定義され、それらが満たす代数的性質も明確でなければならない．例えば以下のような最適化を行なって良いのかどうかが、その中間言語の仕様から決定できなければならない．

```
a*a + 2*a*b + b*b   ->   (a+b)*(a+b)
```

C 言語の仕様書[14] では、整数の表現が 2 の補数表示、1 の補数表示、及び符号と絶対値のいずれの表現をもつマシンでも実装出来るような配慮がされているが、逆に言えば C 言語の演算で成立する代数的性質を制限している．例えば上の最適化を行なってよいかどうかは C 言語の仕様書だけからは決定出来ない（処理系依存）．

2 の補数表現は数学的に最もすぐれた表現であり、1 ワードの整数の集合は単位可換環をなす．よって例えば上の最適化を行なうことが出来る．現在使われている計算機は全て（筆者の知る限り）2 の補数表現を使っていることを考慮し、我々の中間言語における整数は 2 の補数表現によるものとした．このように LIR の設計においては現在使われているハードを考慮し、いくつか特定の仮定を置いている．メモリについても、1 バイトが 8 ビットであるバイトアドレスマシンのみを前提としている．

高級言語の意味論ではデータのドメインはさまざまな型の直和で表現されるが、コンパイラの中間言語が表現するものは実際のハードであり、我々はタグ付きアーキテクチャのような特殊なものは対象外としている．従ってデータを表すドメインは単なるビット列とする．また、メモリでアドレス指定される最小単位のビット列としてバイトを定義する．

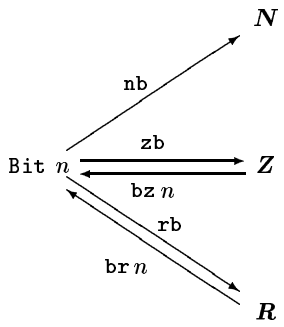
$$
\begin{aligned}
\mathtt{Bit} &: \boldsymbol{N} \to \boldsymbol{Type} \\
\mathtt{Bit}\,n &\triangleq \{0,1\}^n \\
\mathtt{Bits} &: \boldsymbol{Type} \\
\mathtt{Bits} &\triangleq \textstyle\bigcup_{n=0}^{\infty} \mathtt{Bit}\,n \\
\# &: \mathtt{Bits} \to \boldsymbol{N} \\
\#\, b_{n-1}\cdots b_0 &\triangleq n \\
\# &: \{\mathtt{Bit}\,n \mid n \in \boldsymbol{N}\} \to \boldsymbol{N} \\
\#\,\mathtt{Bit}\,n &\triangleq n \\
\mathtt{Byte} &: \boldsymbol{Type} \\
\mathtt{Byte} &\triangleq \mathtt{Bit}\,8
\end{aligned}
$$

以下はビット列を数学的な数と解釈する関数である．自然数の集合 $\boldsymbol{N}$ は理想的な符号なし整数、整数の集合 $\boldsymbol{Z}$ は理想的な符号つき整数、実数の集合 $\boldsymbol{R}$ は理想的な浮動小数と考えている．

$$
\begin{aligned}
\mathtt{nb} &: \mathtt{Bits} \to \boldsymbol{N} \\
\mathtt{nb}\ b_{n-1}\cdots b_0 &\triangleq \textstyle\sum_{i=0}^{n-1} b_i * 2^i \\
\mathtt{zb} &: \mathtt{Bits} \to \boldsymbol{Z} \\
\mathtt{zb}\ b_{n-1}\cdots b_0 &\triangleq \mathtt{nb}\ b_{n-1}\cdots b_0\ - b_{n-1} * 2^n \\
\mathtt{rb} &: \mathtt{Bits} \to \boldsymbol{R}_{\perp} \\
\mathtt{rb}\,b &\triangleq b \text{ の IEEE 浮動小数解釈}
\end{aligned}
$$

ここで zb は 2 の補数表示を仮定した定義になっている．$\mathtt{rb}\,b$ で $b \in \mathtt{Bits}$ が IEEE 解釈で実数とならない場合 $\mathtt{rb}\,b = \perp$ である．以下は数学的な数をビット列に変換する関数である．br も rb 同様部分関数であり、$\mathtt{br}\,n\,x$ で x が IEEE 解釈の n ビット表現を持たない場合は $\mathtt{br}\,n\,x = \perp$ である．

$$\begin{aligned}
\mathtt{bz} &: n{:}\boldsymbol{N} \to \boldsymbol{Z} \to \mathtt{Bit}\,n \\
\mathtt{bz}\,n\,x &\triangleq !b \in \mathtt{Bit}\,n \;\; x \equiv \mathtt{nb}\,b \pmod{2^n} \\
\mathtt{br} &: n{:}\boldsymbol{N} \to \boldsymbol{R} \to \mathtt{Bit}\,n_\perp \\
\mathtt{br}\,n\,x &\triangleq x \text{ の } n \text{ ビット IEEE 浮動小数解釈}
\end{aligned}$$



例えば加算命令は変換関数と数学的な加算演算 $+$ により、以下のように定義される．ここで σ はメモリ、ρ は環境を表す．説明は付録の意味定義を参照のこと．

$$\begin{aligned}
&\mathcal{E}[\![(\mathtt{ADD\ I32}\ x_1\ x_2)]\!]\rho\sigma \\
&\quad\triangleq (\sigma_2, \mathtt{bz}\,32\,(\mathtt{zb}\,v_1 + \mathtt{zb}\,v_2)) \\
&\qquad \textbf{where} \\
&\qquad (\sigma_1, v_1) \triangleq \mathcal{E}[\![x_1]\!]\rho\sigma \\
&\qquad (\sigma_2, v_2) \triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1
\end{aligned}$$

先にふれたとおり、n ビットの 2 の補数表現は数学的にいうと $\boldsymbol{Z}$ の剰余環として得られる単位可換環 $\boldsymbol{Z}/(2^n)$ の代数に対応した、理論的に正しい表現である．ここではビット列 $b \in \mathtt{Bit}\,n$ と $\boldsymbol{Z}/(2^n)$ の剰余類が一対一に対応する．その対応を与えるのが $\mathtt{nb}, \mathtt{zb}, \mathtt{bz}$ であり、b に対して $\mathtt{nb}\,b$ を要素とする $\boldsymbol{Z}/(2^n)$ の剰余類が対応する．言い換えると $\mathtt{nb}\,b$ はその剰余類から代表元 x を $0 \le x < 2^n$ の範囲で選ぶ関数に他ならない．$\mathtt{zb}\,b$ も同様だが代表元 x を $-2^{n-1} \le x < 2^{n-1}$ の範囲で選んでいる点が異なる．$\mathtt{bz}$ は整数 x に、それを要素とする剰余類を表すビット列を対応させている．これらの事実を関数の間に成り立つ定理として述べると以下のようになる．3. は 2. と 4. から導けるが、応用上便利なので上げてある．これらは整数演算式の最適化などで式変形の正当性を証明する際に役に立つ．

$$\begin{aligned}
\mathtt{nb}\,b &\equiv \mathtt{zb}\,b \pmod{2^n} \\
\mathtt{bz}\,n\,(\mathtt{zb}\,b) &= b \\
\mathtt{zb}(\mathtt{bz}\,n\,x) &\equiv x \pmod{2^n} \\
\mathtt{bz}\,n\,x = \mathtt{bz}\,n\,y &\leftrightarrow x \equiv y \pmod{2^n}
\end{aligned}$$

LIR では整数演算でオーバーフローを無視している．その理由はまず第一に COINS で対象とする言語の中で、C と Java はオーバーフローを無視している (直接書かれた定数についてはチェックしているが) という理由が上げられる．第二にオーバーフローを無視する LIR の整数演算は 2 の補数表示により副作用を無視すれば単位可換環をなすが、オーバーフローを無視しないと、そういう数学的に奇麗な性質は成り立たなくなる．すなわち環の演算のみからなる式は数学的に許される自由な変形をおこなっても意味は変わらない．したがってもし「コンパイラは入力がオーバーフローや未定義値を発生させないと仮定し、その範囲で最適化変換を行なって良い」という前提が許されるならば、効率的な最適化を安心して行なうことが可能となる．この前提は少なくとも COINS が対象とする言語では事実上認められていると思われる．

3. シフト演算

幾つかの現実的な仮定を置くことにより、LIR の命令は簡潔なものになったが、シフト演算についてはやや複雑な定義とせざるをえなかった．例えば左シフト命令 `x<<n` において、`n` が負の場合右シフトになるようなマシン (e.g. VAX) もあれば、そうならないマシン (e.g. SPARC) もある．よって C 言語では `n` が負の場合の値は未定義と定めている．すなわち VAX のシフト命令を仮定したような C のプログラムはポータブルではない．

実際にはそのようなプログラムを書くことは少ないかも知れないが、`n` が正の場合でも VAX では `1<<32 = 0` となり、一方 SPARC では `1<<32 = 1` となるので VAX から SPARC へ来た人は要注意である．SPARC の左シフト命令は `n` の下位 5 ビットのみを符号なし整数と解釈し、左シフトの量としているために、このような結果となる．

また右シフト命令 `x>>n` には符号ビットをコピーする算術シフトとコピーしない論理シフトがあり、その区別も必要である．これらシフト命令のバリエーションを整理することにより、LIR のシフト命令は左シフトの算術版 `LSHS` と論理版 `LSHU` 及びは右シフトの算術版 `RSHS` と論理版 `RSHU` のよっつとし、以下のような定義とした（これは算術左シフト）．

$\mathcal{E}[\![(\texttt{LSHS}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\sigma$
$\triangleq (\sigma_2, \texttt{genericshift}\ w\ (\texttt{zb}\ v_1)$
$(\texttt{shiftcount}\ v_2\ [\![m]\!]))$
$\texttt{genericshift}\colon w\colon \texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\ w$
$\texttt{genericshift}\ w\ n\ c \triangleq \texttt{bz}\ w\ (\texttt{floor}(n * 2^c))$
$\texttt{shiftcount}\colon \texttt{Bits} \to \texttt{ShiftModifier} \to \boldsymbol{Z}_\perp$
$\texttt{shiftcount}\ b\ [\![(s\ n\ d)]\!]$
$\triangleq$ **if** $c_0 = c_1$ **then** c_0 **else**
case $[\![d]\!]$ **of** $\texttt{D} \Rightarrow c_1\ \texttt{U} \Rightarrow \perp$
where
$f \triangleq$ **case** $[\![s]\!]$ **of** $\texttt{S} \Rightarrow \texttt{zb}\ \texttt{U} \Rightarrow \texttt{nb}$
$c_0 \triangleq f\ b$
$c_1 \triangleq f(\texttt{bz}\ n\ c_0)$

記号 & は LIR では命令の修飾を表す記号であり、それにつづく S 式をモディファイアという．x_2 の値とモディファイア $m = (s\ n\ d)$ に従い、以下のようにシフトカウントが決まる．まず x_2 の値を n ビットの符号つき $(s = \texttt{S})$、あるいは符号なし $(s = \texttt{U})$ 整数と解釈し、仮のシフトカウントとする．つぎに x_2 の値がその n ビット符号つき (あるいは符号なし) 数として表現出来るなら仮のカウントを正式なシフトカウントとする．そうでない場合、その挙動は d に依存し、$d = \texttt{D}$ なら表現出来なくてもその解釈された値をシフトカウントとする．$d = \texttt{U}$ ならシフトカウントを $\perp$ とし、シフト結果全体を $\perp$ とする．

以下の表はシフトモディファイアと対応するマシンの例である．

S8D	VAX, V60/70
U5D	SPARC(32), MIPS(32), Intel x86
U6D	SPARC(64), MIPS(64), PowerPC(32)
U7D	PowerPC(64)
U64D	Intel MMX

また、結果が未定義になる例として、Intel x86 の SHLD、SHRD で、8、16 ビットレジスタをシフトするときで、シフトカウントの下位 5 ビットが 8、16 を超える場合が上げられる．

4. volatile オブジェクト

マシンはメモリを介して外部とのやりとりを行なう．従って、そのポートとして使われる変数についての最適化は制限される．C 言語ではそのような変数、一般にはオブジェクトを volatile という一種の型により表現することが出来る．LIR では & V というモディファイアを伴った MEM 式により volatile オブジェクトを表す．このセクションでは volatile オブジェクトの意味定義を解説し、またその定義が自然なものであることを示す．

4.1 ランダムステートによる意味付け

ここで v1,v2 がなんらかの装置とのポートとして使われている volatile 変数と仮定し、どのような変換が許されないかを考えてみる．まず明らかに以下のような変換は禁止しなければならない．これはパイプラインスケジューラなどがやるような最適化変換である．

```
t1 = v1        ×        v2 = 1
v2 = 1       ——→        t1 = v1
```

Rule 1: 順序を変えてはならない

以下も同様に禁止しなければならない．これは ded code elimination で行なわれるような最適化変換である．

```
v1 = 1         ×
v1 = 2       ——→        v1 = 2
```

Rule 2: 重複を省いてはならない

装置が volatile オブジェクトを読むことにより動作し、また外部からそこへの書き込まれる可能性もあるので以下の変換も禁止される．これは共通部分式と式の簡略化で行なわれるような最適化変換である．

```
t1 = v1        ×        junk = v1
t2 = v1      ——→        junk = v1
t3 = t1-t2                t3 = 0
```

Rule 3: 値はいつも違う

よって、volatile オブジェクトの「意味」としては、以上のような変換がその意味を変えてしまうような定義をしなければならない．そのために、意味定義におけるメモリーには以下のようなふたつのフィールド、トレース及びランダムステートが存在し、トレースにより Rule 1,2 を、ランダムステートにより Rule 3 を反映させている．

$\texttt{Mem} \triangleq \{\cdots, \texttt{tr}\colon \texttt{Trace}, \texttt{rs}\colon \boldsymbol{R}\}$

ここにトレースはアクションのリストである．また新たなランダムステートを求めるための関数 random があるとする．

$\texttt{Trace} \triangleq [\texttt{Action}]$
$\texttt{Action} \triangleq \texttt{Symbol} \times \texttt{Ltype} \times [\texttt{Bits}]$
$\texttt{random} \ : \ t\colon \texttt{Ltype} \to \texttt{Mem} \to \texttt{Mem} \times \mathcal{T}[\![t]\!]$

処理	Action
読み込み	(READ, $ltype$, [$address$])
書き込み	(WRITE, $ltype$, [$address, value$])

そして volatile メモリーの参照ではそのアドレスがトレースに記録され、値はランダムステートとなる.

$$
\begin{array}{l}
\mathcal{E}[\![(\mathtt{MEM}\ t\ x_1\ \&\ m)]\!] \\
\triangleq \\
\textbf{case}\ [\![m]\!]\ \textbf{of} \\
\quad \mathtt{N} \Rightarrow (\sigma_1, \mathtt{dmread}\,\sigma_1\,(\mathtt{nb}\,v_1)\,[\![t]\!]) \\
\quad \mathtt{V} \Rightarrow \mathtt{random}[\![t]\!]\,(\mathtt{addtotr}\,\sigma_1\,\mathtt{READ}\,[\![t]\!]\,[v_1]) \\
\quad \textbf{where} \quad (\sigma_1, v_1) \triangleq \mathcal{E}[\![x_1]\!]\rho\sigma
\end{array}
$$

また volatile メモリーへの書き込みではそのアドレスと値がトレースに記録される.

$$
\begin{array}{l}
\mathcal{E}[\![(\mathtt{SET}\ t\ (\mathtt{MEM}\ t'\ x_1\ \&\ m)\ x_2)]\!] \triangleq (\sigma_4, v_2) \\
\textbf{where} \\
(\sigma_1, v_1) \triangleq \mathcal{E}[\![x_1]\!]\rho\sigma \\
(\sigma_2, v_2) \triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1 \\
\sigma_3 \triangleq \mathtt{dmwrite}\,\sigma_2(\mathtt{nb}\,v_1)\,[\![t]\!]\,v_2 \\
\sigma_4 \triangleq \textbf{case}\ [\![m]\!]\ \textbf{of} \\
\qquad \mathtt{N} \Rightarrow \sigma_3 \\
\qquad \mathtt{V} \Rightarrow \mathtt{addtotr}\,\sigma_3\,\mathtt{WRITE}[\![t]\!][v_1, v_2]
\end{array}
$$

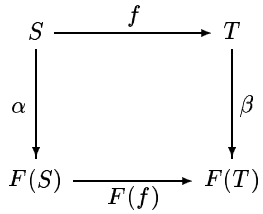
4.2 意味定義の妥当性

このトレースとランダムステートに基づいた意味定義は Rule 1 から Rule 3 にのみ基づいたものであり、その妥当性に疑問がある. 何か volatile 変数のより自然なモデルからの裏付がほしい. LIR の実行可能なプログラム（以下 L プログラム）は volatile 変数をチャネルとして動作するプロセスと見なせる. ここではこの視点で我々の意味定義の妥当性を議論する.

はじめに、bisimulation[21] に基づいた L プログラムの等価性を定義する. その定式化は coalgebra[23][22] で行なう. 次に我々が与えた表示的意味、すなわちトレースとランダムステートに基づく volatile 変数の意味に従った L プログラムの第二の等価性を定義する. 最初の等価性を coalgebra 等価性、第二の等価性をトレース等価性という. そして、このふたつの等価性は同等であることを証明する.

まず以下の議論で使用する概念を要約する. 必要なら bisimulation については[21]、coalgebra については[23]、カテゴリー理論については[19][20] 等を参照されたい.

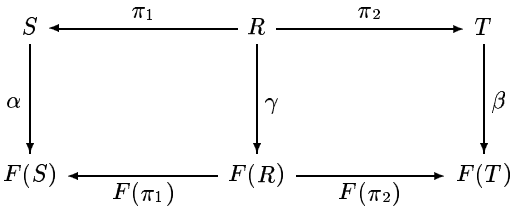
$F{:}\mathcal{C} \to \mathcal{C}$ をカテゴリー $\mathcal{C}$ 上のファンクターとするとき、F-coalgebra とはオブジェクト S と射 $\alpha{:}S \to F(S)$ の組 (S, α) である. このとき S をその coalgebra のキャリアと呼ぶ. ファンクター F が明らかな場合、$F-$ を略し単に coalgebra とも呼ぶ. 以下を可換にする $f{:}S \to T$ を (S, α) から (T, β) への F-homomorphism という.



与えられたカテゴリー $\mathcal{C}$ とファンクター F について、F-coalgebra をオブジェクト、F-homomorphism を射とするカテゴリーを $\mathcal{C}_F$ で表す. このとき、$\mathcal{C}$ を $\mathcal{C}_F$ の underlying category という. $\mathcal{C}_F$ において以下の (R, γ) を (S, α) と (T, β) の F-bisimulation という. またこのような (R, γ) が存在するとき、(S, α) と (T, β) は bisimilar であるという.

$$(S, \alpha) \longleftarrow (R, \gamma) \longrightarrow (T, \beta)$$

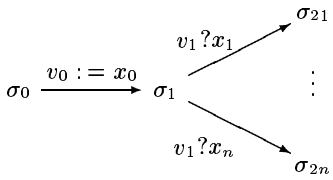
以下の議論で underlying category は $\mathcal{S}et$ である. この場合、上の条件はは以下の図式の可換性と同一である. ここに $R \subset S \times T$、$\pi_1{:}(s,t) \mapsto s$、$\pi_2{:}(s,t) \mapsto t$.



4.3 coalgebra 等価性

coalgebra 等価性は L プログラムのプロセスを以下に述べる方法でラベル付き遷移系と見なした時の bisimulation 等価性である.

我々の意味定義におけるメモリ $\sigma \in \mathtt{Mem}$ はプログラムメモリとデータメモリを含んでいるので、σ をプログラムとも呼ぶ. 以下の遷移図における $\sigma_i \in \mathtt{Mem}$ は全て同一のプログラムであり、遷移によってデータメモリのみが変化している.



ラベル $v := x$ は volatile 変数 v へ値 x を書き込むことによる遷移を意味する. ラベル $v?x_i$ は変数 v の値が x_i である場合の遷移を意味する. いずれの遷移も決定的である. volatile 変数は外部から書き込みが起こり得るが、それは $v?x_i$ による遷移の「直前」

に起こると考える．また、以後の議論では各 LIR 命令（以下 L 式）は高々一つの volatile アクセス（読み書き）を持つと仮定する．LIR では部分式の評価順序が定められているので、これらの仮定は本質的な制限にはならない．

L プログラムを以上のようなラベル付き遷移系と見なすと、bisimulation によりその自然な等価性が定義される．我々はこの等価性をフォーマライズの容易さのために coalgebra により表現する．上記のラベル付き遷移系を coalgebra で表すために、ファンクター F、キャリア X、射 $\alpha: X \to F(X)$ を以下のように定義する．

$$
\begin{aligned}
F &: \text{Functor on the categeory } \mathcal{S}et \\
F(X) &\triangleq \mathtt{Vol} \times \mathtt{Bits} \times X \\
&\quad + \mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, X) \\
&\quad + \{\bot\} \\
\mathtt{Vol} &\triangleq \text{Set of volatile addresses} \\
\mathtt{Memc} &\triangleq \mathtt{Mem} \text{ without } .\mathtt{tr} \text{ and } .\mathtt{rs} \\
\mathcal{S}c &: \mathtt{Memc} \to \mathtt{Memc} \\
\alpha &: \mathtt{Memc} \to F(\mathtt{Memc})
\end{aligned}
$$

ここに $\mathtt{Vol} \subset \boldsymbol{N}$ は volatile オブジェクトのアドレスの集合である．ファンクター F は三つのファンクターの直和であり、最初のものが $v := x$、第二が $v?x$ に、そして $\bot$ は volatile オブジェクトにアクセスしないまま動き続けるプロセスを意味する．coalgebra 等価性ではトレースとランダムステートは参照しないので、それらを $\mathtt{Mem}$ から除いたものがキャリア $\mathtt{Memc}$ である．$\mathcal{S}c$ は LIR の仕様（付録）で定義されている意味関数 $\mathcal{S}$ をトレースとランダムステートを無視するように修正したものである．以下の議論では環境 $\mathtt{Env}$ は本質的ではないので、それも除外している．α はこの $\mathcal{S}c$ により volatile アクセス単位で以下のような実行をする．まず volatile アクセスの直前まで実行を進め、そのアクセスが $x \in \mathtt{Bits}$ を $v \in \mathtt{Vol}$ で示されるオブジェクトへ書き込む命令ならば α は (v, x, σ') を返す．ここに σ' は書き込み後のメモリである．またもしその volatile アクセスが $v \in \mathtt{Vol}$ からの読み込みの場合は $(v, f{:}\mathtt{Bits} \to \mathtt{Memc})$ を返す．ここに $f\,x_i$ は先のラベル付き遷移系において、ラベル $v?x_i$ による遷移先の σ である．

以上で我々は LIR におけるプログラム全体の coalgebra を定義したことになる．特定のプログラム σ の coalgebra はその部分 coalgebra $(\overline{\sigma}, \alpha)$ であり、そのキャリア $\overline{\sigma}$ は以下のように定義される．

$$
\begin{aligned}
\overline{\sigma} &\triangleq \{\sigma' \in \mathtt{Memc} \mid \sigma \xrightarrow{\alpha^*} \sigma'\} \\
\sigma \xrightarrow{\alpha^*} \sigma' &\triangleq \sigma = \sigma' \vee \\
&\quad \exists \sigma''\ \sigma \xrightarrow{\alpha^*} \sigma'' \wedge \sigma'' \xrightarrow{\alpha} \sigma' \\
\sigma \xrightarrow{\alpha} \sigma' &\triangleq \alpha\,\sigma \in \mathtt{Vol} \times \mathtt{Bits} \times \mathtt{Memc} \\
&\quad \wedge \alpha\,\sigma\,!\,2 = \sigma' \\
&\quad \vee \\
&\quad \alpha\,\sigma \in \mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, \mathtt{Memc}) \\
&\quad \wedge \exists b\ (\alpha\,\sigma\,!\,1)b = \sigma'
\end{aligned}
$$

以上より coalgebra 等価性を次のように定義する．

> プログラム $\sigma_1, \sigma_2 \in \mathtt{Memc}$ から作られる部分 coalgebra $(\overline{\sigma_1}, \alpha)$ と $(\overline{\sigma_2}, \alpha)$ が bisimilar のとき、σ_1 と σ_2 は coalgebra 等価であるといい、$\sigma_1 \stackrel{c}{=} \sigma_2$ と表す．

この等価性の定義は OS のように停止しないで動き続けるプログラムの等価性も含んだ自然なものである．一方我々が与えた LIR の表示的意味では停止しないプログラムは全て $\bot$ という扱いになってしまう．しかしプログラムが停止する場合には、これらの二つの等価性は等価であることが示される．

4.4 トレース等価性

はじめに LIR の本来の意味論すなわちトレースとランダムステートに基づいたプログラムの等価性（トレース等価性）を定義する必要がある．そのために、プログラムはある特定のメイン関数から開始するものとする．まず以下の定義をおこなう．

$$
\begin{aligned}
\beta &: \mathtt{Mem} \to \mathtt{Mem}_\bot \\
\beta\,\tau &\triangleq \text{execute until the next volatile} \\
\mathtt{t} &: \mathtt{Mem} \to \boldsymbol{R} \to \mathtt{Trace} \\
\mathtt{t}\,\tau\,r &\triangleq [a_1, \ldots, a_n] \\
\mathtt{c} &: \mathtt{Mem} \to \mathtt{Memc} \\
\mathtt{c}\,\tau &\triangleq \tau \text{ without random states and traces}
\end{aligned}
$$

ここに β は α と同様の volatile アクセス単位での実行関数であるが、これは本来のトレースとランダムステートを使った意味関数 $\mathcal{E}$ と $\mathcal{S}$ に基づくものである．関数 $\mathtt{t}$ はプログラム τ の全体の挙動としてのトレースを返す．引数 r は τ の実行におけるランダムステートの初期値である．関数 $\mathtt{c}$ はプログラム τ からランダムステートとトレースを無視（削除）するものである．

プログラム τ の挙動は乱数関数 `random` 及びランダムステートの初期値 $r \in \boldsymbol{R}$ で完全に決定される．しかし `random` の実現の仕方が高々加算であることから、ある特別な `random` 関数を fix し、可能なあらゆ

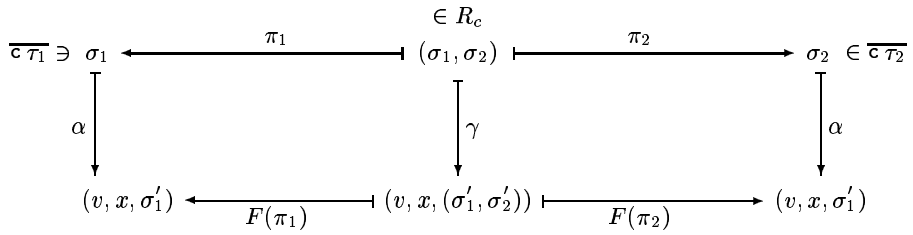


図 1 $\mathtt{Vol} \times \mathtt{Bits} \times (-)$ の場合

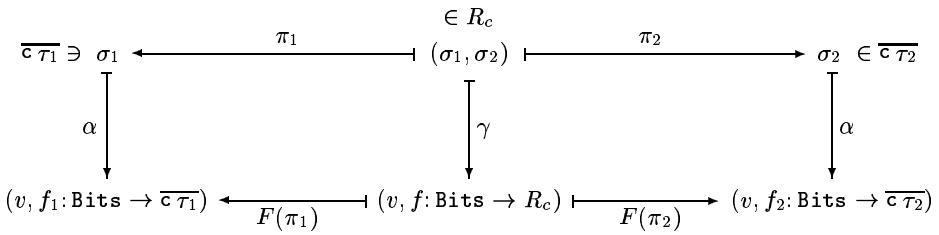


図 2 $\mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, -)$ の場合

るランダム列を初期値の違いでのみ表現することは可能である．よって以後プログラムの挙動は単に初期値 r で完全に決定されるとして議論する．以下がトレース等価性の定義である．

> プログラム $\tau_1, \tau_2 \in \mathtt{Mem}$ が以下を満たす時、
> $\forall r \in \boldsymbol{R}\ \ \mathtt{t}\,\tau_1\, r = \mathtt{t}\,\tau_2\, r$
> それらはトレース等価であるといい、 $\tau_1 \overset{t}{=} \tau_2$ と表す．

4.5 二つの等価性の同値性

以下の定理は coalgebra 等価性とトレース等価性が同値であるということを意味する．証明はスケッチである．

定理 ($\overset{t}{=}$ と $\overset{c}{=}$ は同値)

$$\forall \tau_1, \tau_2 \in \mathtt{Mem}\ \ \tau_1 \overset{t}{=} \tau_2 \leftrightarrow \mathtt{c}\,\tau_1 \overset{c}{=} \mathtt{c}\,\tau_2$$

証明

プログラム $\tau_1, \tau_2 \in \mathtt{Mem}$ が $\tau_1 \overset{t}{=} \tau_2$ を満たすと仮定し、以下を定義する．

$$\begin{aligned} R_t &\overset{\triangle}{=} \{(\tau_1', \tau_2') \in \mathtt{Mem}^2 \mid \\ &\qquad \exists r \in \boldsymbol{R} \\ &\qquad (\tau_i := \{\mathtt{rs} = r\}) \xrightarrow{\beta^*} \tau_i'\ (i = 1, 2)\} \\ R_c &\overset{\triangle}{=} \{(\mathtt{c}\,\tau_1, \mathtt{c}\,\tau_2) \in \mathtt{Memc}^2 \mid (\tau_1, \tau_2) \in R_t\} \end{aligned}$$

遷移関数の定義をフォーマルには与えていないが、$(\tau_1, \tau_2) \in R_t$ ならば $\beta \colon \tau_1 \to \tau_1'$ と $\beta \colon \tau_2 \to \tau_2'$ が同じアクション、すなわち供に書き込みか読み込みか $\perp$ であることは明らかである．

以下 R_c が $(\overline{\mathtt{c}\,\tau_1}, \alpha)$ と $(\overline{\mathtt{c}\,\tau_2}, \alpha)$ のある bisimulation であることを F の場合分けで示す．

1. $(\mathtt{Vol} \times \mathtt{Bits} \times (-))$ の場合:

図 **1** 参照．R_c の定義より、ある $\tau_1', \tau_2' \in R_t$ が存在して $\sigma_i = \mathtt{c}\,\tau_i'\ \ (i = 1, 2)$ となる．今の場合、τ_i の選択は任意である．

R_t の定義より $\tau_1', \tau_2' \in R_t$ のアクションは等しい．例えばそれが $v := x$ であれば、以下が成り立つ．

$$\alpha \colon \mathtt{c}\,\tau_i' \mapsto (v, x, \mathtt{c}\,\tau_i'')\ \ (i = 1, 2)$$

すなわち図 1 の下の左右は正しい．よってもし $(\sigma_1', \sigma_2') \in R_c$ であれば、 γ を図式どうりに定義できるが、上の議論より $(\sigma_1', \sigma_2') = (\mathtt{c}\,\tau_1'', \mathtt{c}\,\tau_2'')$ であり R_t の定義から $(\tau_1'', \tau_2'') \in R_t$ よって $(\sigma_1', \sigma_2') \in R_c$ であり、最初の場合は示された．

2. $(\mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, -))$ の場合:

同様の議論により 図 **2** で示されていることはほぼ明らかである．証明を完結するには γ の定義が必要である．もし任意の $x \in \mathtt{Bits}$ について $(f_1\, x, f_2\, x) \in R_c$ であれば、 $f \overset{\triangle}{=} f_1 \times f_2$ として、γ が定義出来る．そのような $x \in \mathtt{Bits}$ を固定し、 $\sigma_i = \mathtt{c}\,\tau_i'$ でかつ $\beta \colon \tau_i' \mapsto \tau_i''$ のアクションが $v?x$ となるように $\tau_1', \tau_2' \in R_t$ を選ぶ．これは R_t の定義及び乱数の初期値のみで全ての挙動が決定するという議論から可能である．よって $f_i\, x = \mathtt{c}\,\tau_i''\ \ (i = 1, 2)$ すなわち $(\tau_1'', \tau_2'') \in R_t$ である． 逆の証明、及び $\perp$ の場合については省略する．

5. 多値関数の導入

関数呼び出しはリターゲッタブルコンパイラで問題になる部分のひとつである．ターゲットマシンを固定したとしても尚、さまざまなコーリングシーケンスがあり得る．引数の一部は通常レジスタに割り当てられるが、それは後にレジスタアロケータが自由に割り当ててよいものではなく、関数呼び出しの規約に従った割り当てをしなければならない．このような特殊な扱いが必要なレジスタを pre-allocated レジスタという．

こういった、いわゆる practical issues は標準的なコンパイラの教科書[1)2)4)] では取り上げられていないが、実際には重要で難しい問題であり、最近でも ML-RISC プロジェクト[10)] で一つの提案[11)] がなされている．彼らの方法は SSA との変換過程で pre-allocated レジスタの処理をするものであり、SSA に特化されているし、また提案されている方法も複雑なので、ここでは言及しない．

このセクションでは以下の C コードを例とし、代表的なアプローチと我々のアプローチを説明する．

```
int f(int a,int b)
{
  int r;
  r = g(a+b,b);
  return r*a;
}
```

以下で使用する中間言語のシンタックスは LIR のものも含めて説明のために簡略化したものである．vn は仮想レジスタ、rn は実レジスタを表すとし、引数は r0,r1 、返却値は r0 に入るようなコーリングコンベンションを仮定する．

5.1 よくある方法

実際的でよくある方法はコーリングシーケンスを表す特殊な式を導入する方法である．

```
v0 = arg(0)
v1 = arg(1)
v2 = v0+v1
setarg(0,v2)
setarg(1,v1)
v3 = call g
v4 = v3*v0
return v4
```

ここに arg(i) は i 番目の f の引数を表す特殊な式と考える．また setarg(i,v) は i 番目の g の引数をセットする特殊な式である．return v は f の返却値を表す．

このアプローチの利点は、コード最適化の段階で pre-allocated レジスタのケアが不要という点である．実際実レジスタは後にレジスタアロケータがそれぞれの特殊な式と、定められたコーリングコンベンションに従って実レジスタを割り当てることになる．

このアプローチの欠点は、arg といった特殊な式に対して、コード最適化ルーチンは特別な注意を払わなければならない点である．それはたいしたコストではない、とも言えるが、より良い表現があればそれに越したことはない． ある意味で、我々のアプローチはこの多少いやな部分を解決したものである．我々のアプローチを紹介するまえに、GCC の思い切った解決法を紹介する．

5.2 GCC の方法

GCC は最初の RTL 生成時においてパターンマッチングを行ない、ターゲットマシン固有の表現を展開する、という戦略を取っている．これはリターゲッタブルコンパイラの中でも GCC 独自の戦略であるらしい[9)]．GCC では、従って先の C コードが最初に RTL へと変換された時点でコーリングコンベンションは完全に展開されている．

```
v0 = r0
v1 = r1
r0 = v0+v1
r1 = v1
r0 = call g, clobber r1
r0 = r0*v0
use r0
```

ここに clobber は関数呼び出し中に破壊されるレジスタを明示するための指定である．use はそのレジスタが使用されていることを明示する式である．このように RTL では一般に pre-allocated レジスタと仮想レジスタが混在している．フレームポインタやスタックポインタも RTL では pre-allocated レジスタとして明示的に現れている．

このアプローチの利点は明らかである．コード最適化はまさにマシン固有の表現を対象に動作するのだから、よりクォリティーの高い最適化を行なうことが可能となる．明らかにこのアプローチでなければやりにくい最適化として、関数呼び出しが連続する場合にスタックの pop をまとめて行なうような最適化（defer pop）が上げられる．

このアプローチの欠点もまた明らかであろう．pre-allocated レジスタをケアしなければならない、というのはコード最適化を複雑なものにしてしまう．また、RTL 生成時に既にコリングコンベンションが展開さ

れるということは RTL ベースでのインライン展開や tail recursion の最適化を困難にする．実際 GCC ではパーサレベルでそのような最適化を行なっており、RTL が生成された以後はそういった最適化は行なわない．

5.3 LIR のアプローチ

最初のアプローチの問題点は、中間言語でも高級言語のように関数インタフェースを保持するようにすることで解決出来る.（以下の LIR コードは説明のためのシンタックスであり正式なものではない）LIR はコンパイラの中間言語であるが、このようなモジュールと関数インタフェースを持っている．

```
module foo
  function f
    reg v0,v1,v2,v3;
  prologue (v0,v1)
  call g (v0+v1,v1) (v2)
  v3 = v2*v0
  epilogue (v3)
```

このモジュールはひとつのコンパイル単位となる．`reg v0,v1,v2,v3` は関数 `f` でのローカルな仮想レジスタである．`prologue, epilogue` は関数の始まりと終りを示し、`prologue` の引数が関数の引数、`epilogue` の引数が返却値である．返却値は複数でもよい．ここで重要なのは `prologue` や `epilogue` は式ではない、ということである．コード最適化ルーチンが扱う式の中には最初のアプローチにおける `arg(i)` や `return` といった特殊なケアが必要となるものは存在しない．

我々のアプローチでは、GCC と異なりコード最適化の段階では一切 pre-allocated レジスタが現れないようにしている．よって、確かに GCC の defer pop のような最適化はやりにくくなるが、一方で LIR を対象としたコード最適のひとつとして、インラン展開、末尾再帰、部分評価などの最適化が可能になる．

LIR ではレジスタの表現自体に仮想レジスタと実レジスタの区別はない．その区別はスコープによってなされる．実レジスタとはグローバルなスコープで宣言されているレジスタと見なす．そしてレジスタアロケーションも含めてコンパイラの全ての処理をプログラム変換として捉えている．レジスタアロケーションとはローカルなレジスタ（仮想レジスタ）をグローバルなものに等価に変換する処理と考える．先の例はレジスタアロケーションの後、以下のように変換される．

```
module foo
  reg r0,r1,r2;

  function f
  prologue (r0,r1)
  r2 = r0
  r0 = r0+r1
  call g () (r0)
  r0 = r0*r2
  epilogue (r0)
```

プログラム変換とはいうまでもなくプログラムの意味を変えない変換である．よってコンパイラの処理をプログラム変換として定義するためにはプログラム、すなわち LIR の意味が明確に定義されている必要がある．

Glasgow の HASKELL コンパイラは GNU C を「ターゲットアセンブラ」としてきたが、別に C-- [12] という中間コードも提案され、ML-RISC [10] を利用してインプリメントされている．これは C 言語をよりアセンブラに近付ける、というアプローチであり、構造体型といった高級な型は排除されているが、代わりに多値関数が導入されている．その理由はインライン展開や末尾再帰の便宜のためである．彼らは C 言語から無駄を削って行って関数インタフェースが残った．我々はアセンブラ言語レベルの中間コードに構造を追加することで、やはり関数インタフェースを保持したほうが良い、という結論に達した．

6. 例

ここでは中間言語 LIR を簡単な例によって紹介し、LIR の全体構成を説明する．図 **3** はふたつのソース `main.c` と `sub.c` からなる C 言語のプログラムであり、`prodv` は `fold1` を利用して配列 `v` の要素の積を計算する．

`main.c` を LIR に変換したものが 図 **4** である．LIR は S 式で表現されおり、セミコロンから行末まではコメントと扱う．左の番号は説明のための参照用であり、 LIR の一部ではない．ここに整数、アドレス、浮動小数ともに 32 ビットで、アラインメントの要求は全て 4 バイト境界とする．またコード、データの置かれるセグメント名はそれぞれ `text,data` とする．

これは LIR のコンパイル単位である L モジュールの例である．L モジュールはモジュール名 (ここではソースファイルに基づいて付けた)、連想リスト、そして関数とデータの定義列から成り立つ．関数定義は関数名、関数のローカルな連想リスト、そして L 式列

```
main.c :   extern float fold1(float f(float,float), float v[], int n);
           static float v[] = {1, 2.5, 3}
           static int n = sizeof v / sizeof v[0];
           static float fmul(float x, float y){float r=x*y; return r;}
           float prodv(){return fold1(fmul,v,n);}

sub.c :    float fold1(float f(float,float), float v[], int n)
           {
             int i; float r;
             for (r=v[0], i=1; i<n; i++) r = f(r,v[i]);
             return r;
           }
```

図 3 C のソースコード

```
 1  (MODULE "main"
 2    (ALIST
 3      ;; 名前  静的    型      境界 セグメント  リンケージ
 4      ("fold1" STATIC UNKNOWN  4  "text"     XREF)
 5      ("v"     STATIC 96       4  "data"     LDEF)
 6      ("n"     STATIC I32      4  "data"     LDEF)
 7      ("fmul"  STATIC UNKNOWN  4  "text"     LDEF)
 8      ("prodv" STATIC UNKNOWN  4  "text"     XDEF))
 9    ;; 関数 fmul の定義
10    (FUNCTION "fmul"
11      (ALIST
12        ;; 名前 フレーム  型  境界  オフセット
13        ("x"    FRAME    F32  4    0)
14        ("y"    FRAME    F32  4    4)
15        ("r"    FRAME    F32  4    8))
16      ;; L 式列
17      (PROLOGUE (12 0) (MEM F32 (FRAME I32 "x")) (MEM F32 (FRAME I32 "y")))
18      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=x*y
19               (MUL F32 (MEM F32 (FRAME I32 "x"))
20                        (MEM F32 (FRAME I32 "y"))))
21      (EPILOGUE (12 0) (MEM F32 (FRAME I32 "r"))))
22    ;; データ v,n の定義
23    (DATA "v" (F32 1.0 2.5 3.0))
24    (DATA "n" (I32 3))
25    ;; 関数 prodv の定義
26    (FUNCTION "prodv"
27      (ALIST
28        ("t1" FRAME F32 4 0)) ; コンパイラが生成した変数
29      (PROLOGUE (4 0))
30      (CALL (STATIC I32 "fold1") ; t1=fold1(fmul,v,n)
31            ((STATIC I32 "fmul") (STATIC I32 "v") (MEM I32 (STATIC I32 "n")))
32            ((MEM F32 (FRAME I32 "t1"))))
33      (EPILOGUE (4 0) (MEM F32 (FRAME I32 "t1")))))
```

図 4 main.c の LIR コード

と呼ばれる命令列から成り立つ．データの定義はデータ名とその内容からなり、内容は具体的な値の他に領域を確保する命令、境界制御命令も含む．

連想リストの各エントリはその第一要素の情報を表している．エントリの解釈はその第二要素に依存する．エントリの第二要素が STATIC なら名前は静的なアドレス (最終的にリンカが解決する) を表す．例えば L モジュール "main" の第 6 行は "n" が静的なアドレスを表し、さらにそこには I32 という型 (32 ビット整数) のオブジェクトが置かれており、境界の要求は 4 バイトで data セグメントに割り当てられ、リンケージはローカル定義 LDEF すなわちモジュール外から参照出来ないということを意味する．リンケージには他に外部に対して定義することを意味する XDEF と、外部参照を意味する XREF がある．

第 6 行の v の型は単に 96 とあるが、これは 96(=32*3) ビットのオブジェクトであることを意味する型である．他に LIR は浮動小数用の型 (例えば第

```
 1  (MODULE "sub"
 2    (ALIST
 3      ("fold1" STATIC UNKNOWN 4 "text" XDEF))
 4    (FUNCTION "fold1"
 5      (ALIST
 6        ("f" FRAME I32 4 0)
 7        ("v" FRAME I32 4 4)
 8        ("n" FRAME I32 4 8)
 9        ("i" FRAME I32 4 12)
10        ("r" FRAME F32 4 16))
11      (PROLOGUE (20 0) (MEM I32 (FRAME I32 "f"))
12                       (MEM I32 (FRAME I32 "v"))
13                       (MEM I32 (FRAME I32 "n")))
14      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=v[0]
15               (MEM F32 (MEM I32 (FRAME I32 "v"))))
16      (SET I32 (MEM I32 (FRAME I32 "i")) ; i=1
17               (INTCONST I32 1))
18      (DEFLABEL "L1")
19      (JUMPC (TSTLT I32 (MEM I32 (FRAME I32 "i")) ; if (i<n) goto L2; else goto L3
20                        (MEM I32 (FRAME I32 "n"))) (LABEL I32 "L2") (LABEL I32 "L3"))
21      (DEFLABEL "L2")
22      (CALL (MEM I32 (FRAME I32 "f")) ; r=f(r,v[i])
23            ((MEM F32 (FRAME I32 "r"))
24             (MEM F32 (ADD I32 (MEM I32 (FRAME I32 "v")) ; v[i]
25                               (MUL I32 (MEM I32 (FRAME I32 "i"))
26                                        (INTCONST I32 4)))))
27            ((MEM F32 (FRAME I32 "r"))))
28      (SET I32 (MEM I32 (FRAME I32 "i")) ; i++
29               (ADD I32 (MEM I32 (FRAME I32 "i"))
30                        (INTCONST I32 1)))
31      (JUMP (LABEL I32 "L1"))
32      (DEFLABEL "L3")
33      (EPILOGUE (20 0) (MEM F32 (FRAME I32 "r")))))
```

図 5 sub.c の LIR コード

13 行にある F32) もある． LIR の型システムはこのようにハードが直接認識出来るものとなっている．これを L タイプという．コンパイル時にその実際の大きさが分からない場合は UNKNOWN としているが、これは L タイプではない．

第 10 行は簡単な関数定義の例である．L 式列の最初と最後の命令 PROLOGUE、EPILOGUE は関数としてのインタフェースを定義している．これは実際のハードでは通常存在しない命令であるが、LIR ではこのようなインタフェースを付加することにより、コード最適化において幾つかの定理を提供している．一般に LIR では関数は多値関数であり、PROLOGUE の第三要素以降が値を受けとる変数のならび、EPILOGUE の第三要素以降が返される多値を表す式のならびである．

PROLOGUE、EPILOGUE の L 式の第二要素 $(w_f\ w_r)$ はフレームのサイズを表す．最初の w_f はフレームのサイズだが、次の w_r はレジスタフレームのサイズである．これらについては後に説明する．

関数定義中の連想リストのエントリ (例えば第 13 行) は、その名前がフレーム中のオフセットを表すものであることを示す FRAME を第二要素として持つ．このエントリでは型、境界、フレーム中のオフセットが要素である．

第 18 行が典型的な L 式である．対応する C 言語のコードと比較されたい．まずメモリーの参照 (読み書き) は明示的に (MEM 型 アドレス) という L 式で表されている点が C 言語と異なる．型はそのメモリー参照が表すオブジェクトの型である．そして、たとえば第 13 行で定義されている "x" は、フレームに割り当てられた変数のアドレスを表す L 式 (フレーム式) により (FRAME I32 "x") と表される．ここで I32 はこの L 式の型すなわちアドレスの型である．

フレームポインタは陽に現れていない．これによりコード最適化フェーズでの処理が容易となる．しかし LIR にはフレームポインタも存在し、フレーム式の意味定義ではそれが参照されている．詳細はここでは述べない．

第 30 行が LIR の関数呼びだし命令であり、多値を返すために (CALL 関数アドレス (引数...) (結果を入れる変数...) というシンタックスとなっている．

関数呼びだしは他の命令の部分にはなれない．よって t1 という一時変数が導入されている．

第 31 行の (MEM I32 (STATIC I32 "n")) はグローバル変数の参照例であり、第 7 行で宣言された "n" がスタティックなアドレスを表す L 式により (STATIC I32 "n") と表されている．

この例で欠けている重要なものにレジスタがある．LIR のレジスタは (REG 型 "regname") というシンタックスで表され、"regname" は連想リストにおいては ("regname" REG 型 オフセット) というエントリで表される．レジスタの名前が表すものはそれが保持するオブジェクトであり、レジスタのアドレスではない．そもそもレジスタのアドレスというものは通常存在しないが、LIR ではレジスタメモリというものを仮定し、各レジスタはそのメモリ中に置かれるとして扱う．上のレジスタのエントリにおけるオフセットとはそのレジスタのアドレスを決定するのに使われる．先に触れたレジスタフレームのサイズ w_r もまたそのアドレスの決定に使われる．詳細はここでは説明しない．

図 **5** は sub.c に対応する LIR コード（L モジュール）である．この関数定義は制御構造を含む例である．いわゆる構造化プログラミング向けの高級な制御構造は提供せず、代わりにいくつかの条件分岐命令を提供する．(DEFLABEL "ラベル") によりラベルを定義し、参照は (LABEL 型 "ラベル") で行なう．この型はコードのアドレスを表す整数の適当な型である．JUMP は無条件分岐、JUMPC は条件分岐である．他に JUMPN という多方向分岐命令もある．

ラベルのスコープは関数内であり、関数の外のラベルへ分岐することは出来ない．むろん、ラベル名は最終的に生成されるアセンブラ言語にそのまま使われるなら、アセンブラでは関数ローカルなラベルというものは存在しないので、適当なリネーム処理が必要とされる．しかし LIR の言語仕様上はあくまでラベルのスコープは関数内と定める．

第 22 行は関数の間接呼びだしの例にもなっている．少なくとも C 言語を表現する必要上、間接呼びだしは必要であり、CALL 式の第一引数はしたがって関数のアドレスを表す任意の式が書けるようになっている．しかし LIR のモデルではこの計算されたアドレスは必ず LIR で書かれた関数であると考える．

7. ま と め

実用規模のコンパイラ中間言語 LIR を設計し、従来はあまり試みられなかった厳密な仕様記述を与えた．また volatile オブジェクトの表示的意味を与え、その妥当性を示した．LIR は現在成功しているリターゲッタブルコンパイラの中間言語、特に GCC の RTL を参考にしつつも、幾つかの新たな試みをしている．

現在 COINS コンパイラは簡単な C のコード生成が出来るようになった段階であり、LIR の設計目標の一つである信頼性のあるコード最適化を容易に実現する、という点で評価が出来る段階には至っていない．また SIMD 最適化の研究[24] において、if 変換のための特別な式、及びコンディションコードの表現が必要になった．これらは正式な仕様として採り入れる予定である．今後の COINS プロジェクトの進展に期待したい．

謝辞 LIR について議論していただいた COINS のメンバーと萩谷研究室の方々に謹んで感謝の意を表する．

参 考 文 献

1) A.V.Aho, R.Sethi, and J.D.Ullman: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.
2) Andrew W.Appel: *Modern Compiler Implementation in ML*, Cambridge Univ. Press, 1998.
3) Steven S.Muchnick: *Advanced Compiler Design and Implemantation*, Morgan Kaufman, 199x.
4) 中田育男: コンパイラの構成と最適化, 朝倉書店, 1999.
5) Richard M.Stallman: *Using and Porting GNU CC*, Free Software Foundation, 1999.
6) W.Clinger, J.Rees (eds): *Revised^4 Report on The Algorithmic Language Scheme*, 1991.
7) James Gosling, et al: The Java Language Specification, Sun Microsystems, 2000 .
8) C.Fraser,D.Hanson: *A Retargetable C Compiler*, The Benjamin/Cummings Publishing Company,Inc.
9) R. Leupers and P. Marwedel: *Retargetable Compiler Technology for Embedded Systems Tools and Applications*, Kluwer Academic Publishers, 2001.
10) L. George: *MLRISC: Customizable and Reusable Code Generators.* Technical report, Bell Laboratories, Murray Hill, 1997.
11) Allen Leung, Lal George: *Static Single Assignment Form for Machine* PLDI 99.
12) Simon P.Jones, et al.: C--: A Portable Assembly Language, *Implementing Functional Languages 1997*, LNCS, 1998.
13) Andrew W.Appel: *Compiling with Continuations*, Cambridge Univ. Press, 1992.

14) ISO: *ISO/IEC 9899:1999 Programming languages - C*, 1999.
15) Milner,R., Tofte,M., and Harper,R.: *The Definition of Standard ML*, MIT Press, Cambridge, MA 1990.
16) Carl A.Gunter: *Semantics of Programming Languages*, MIT Press, 1992.
17) L. Allison: *A Practical Introduction to Denotational Semantics*, Cambridge Univ. Press, 1986.
18) J.E.Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, The MIT Press, 1977.
19) B.C.Pierce: *Basic Category Theory for Computer Scientists*, MIT Press, 1991.
20) S.Mac Lane: *Categories for the Working Mathematician*, Springer-Verlag, 1971.
21) R.Milner: *A Calculus of Communicating Systems*, LNCS 92, 1980.
22) P.Aczel and N.Mendler: A final coalgebra theorem, In *Proceedings Category Theory and Computer Science*, LNCS 389:357–365, 1989.
23) J.J.M.M.Rutten: Universal coalgebra: a theory of systems, *Theoretical Computer Science*, 249:3–80,2000.
24) 藤波 順久, 阿部 正佳: SIMD 型拡張命令をもっと使った最適化への道のり, 第 43 回プログラミング・シンポジウム報告集, 2002.

付録 LIR の仕様

A.1 準 備

定義を意味する記号は 「 $\triangleq$ 」を使う. 例えば $\mathtt{foo} \triangleq \{1, 2, \ldots, 100\}$ は foo を 1 から 100 までの自然数の集合であると定義している. このように、常識的に判断して誤解がない範囲で、表記の簡潔さのために「...」を使用する.

論理記号は $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, $\exists!$ を使う. $\exists! x \in X\ P(x)$ は $P(x)$ である $x \in X$ がただ一つ存在することを意味する. ことのきその一意的に存在する x を $!x \in X\ P(x)$ で表す.

論理記号の構文的な結合順位は最初に提示した順に低くなるとする. 他の集合や演算の記号についても一応はこのように結合順位を定めるが、誤解の恐れがある場合には冗長な括弧により誤解の無い記述を行なう.

集合論の表記 $\emptyset, \{x_1, \ldots, x_n\}$, $\{x \in X \mid P(x)\}$, A^B, $\times$, $\cup$, $\cap$, $+$, $-$, $!$, $\subset$, $\in$ を使う. 二項記号の結合順位はこの順で低くなるとする. $\emptyset$ は空集合を表す. 集合の包含関係「$\subset$」は広義、すなわち $A \subset B \triangleq \forall x \in A\ x \in B$ である. $A \times B$ は直積、A^B は配置集合であり、B が自然数 n の場合は A の n 直積 $A \times \cdots \times A$ とする. 直積 $A_1 \times \cdots \times A_n$ の要素は $x_i \in A_i$ として $(x_1, \ldots, x_n)$ で表す.

本ドキュメントにおいて直積の要素や後に定義するリストを含め、一般にある列の要素を特定する際、「n 番目」という表現は 0 オリジン $(0 \leq n)$、「第 k」という表現は 1 オリジン $(1 \leq k)$ とする.

直積の要素 $x = (x_0, x_1, \ldots)$ の i 番目の要素 x_i を $x!i$ で表す. $A_1 + \cdots + \mathcal{A}_n$ は内部直和である. すなわちそれは $A_1 + \cdots + A_n = A_1 \cup \cdots \cup A_n$ でかつ $i \neq j \rightarrow A_i \cap A_j = \emptyset$ なる集合である. 外部直和は使用しない. $A - B$ は集合の差、すなわち $A - B \triangleq \{x \in A \mid x \notin B\}$ である. $\cup, \cap$ は $\cup_{i=1}^n S_i$, $\cap_{i=1}^n S_i$ という形でも使用する. ($\cup_{i=1}^n S_i \triangleq S_1 \cup \cdots \cup S_n$ である. $\cap_{i=1}^n S_i$ についても同様である.)

本ドキュメントでは以下の型及び型構成子を使用する. 構成子の結合順位はこの提示順に低くなるとする. 直積、直和、差は集合のそれと同じであるが、構成子全体の結合順位明記のために加えてある. 型は単純に集合と扱う. よって型の等価性は集合のそれである. 型と呼ぶか集合と呼ぶかは単に気分の問題である. 同様に $a{:}X$ というシグネチャは $a \in X$ と同じであり、使い分けに深い意味は無い.

$\boldsymbol{R}$ は実数の集合であるが、通常の意味で順序体 $(\boldsymbol{R}, \leq)$ とも見なす. 一般に空でない全順序集合 $(X, \leq)$ について、その最大値を $\mathtt{max}\, X$、最小値を $\mathtt{min}\, X$ で表す. $\boldsymbol{R}$ の体としての四則演算は $*, /, +, -$ で表す. 結合順位は通常通りである. また $-$ は単項のマイナスの意味でも使い、結合順位は二項演算より高い. $\sum$ は総和記号を表す. $x, y \in \boldsymbol{R}$, $0 < x$ について x^y で冪乗を表す.

これらの演算は $\boldsymbol{R}$ の部分集合にも適用する. 例えば $a, b \in \boldsymbol{Z}$ について、a/b は $a, b \in \boldsymbol{R}$ と見なした演算であり、$a/b \in \boldsymbol{R}$ である.

本ドキュメントでは乗算記号は省略しない. したがって ab は a と b の積ではなく、ab という記号であるが、ラムダ記法の束縛変数は例外である (後述).

$$
\begin{aligned}
\boldsymbol{N} &\triangleq 0\text{ 以上の整数の集合}\\
\boldsymbol{Z} &\triangleq \text{整数の集合}\\
\boldsymbol{R} &\triangleq \text{実数の集合}\\
\boldsymbol{Bool} &\triangleq \{\mathbf{true}, \mathbf{false}\}\\
\boldsymbol{Type} &\triangleq \text{型であることを表す}\\
[\tau] &\triangleq \tau\text{ のリスト}\\
\{l_1{:}\,\tau_1, \ldots, l_n{:}\,\tau_n\} &\triangleq \text{レコード型}\\
\tau_1 \times \cdots \times \tau_n &\triangleq \tau_1, \ldots, \tau_n\text{ の直積}\\
\tau_1 + \cdots + \tau_n &\triangleq \tau_1, \ldots, \tau_n\text{ の直和}\\
\tau_1 - \tau_2 &\triangleq \tau_1\text{ と }\tau_2\text{ の差}\\
\alpha \to \beta &\triangleq \alpha\text{ から }\beta\text{ への関数型}\\
a{:}\,\alpha \to \beta(a) &\triangleq \text{依存型 } (\beta{:}\,\alpha \to \boldsymbol{Type})
\end{aligned}
$$

整数に関してガウスの合同式 $x \equiv y \pmod m$ を使用する．実数を整数に変換する以下の関数を使用する．

$$
\begin{aligned}
&\mathtt{floor}, \mathtt{ceiling}, \mathtt{truncate}{:}\ \boldsymbol{R} \to \boldsymbol{Z}\\
&\mathtt{floor}\, x \triangleq \mathtt{max}\{z \in \boldsymbol{Z} \mid z \le x\}\\
&\mathtt{ceiling}\, x \triangleq \mathtt{min}\{z \in \boldsymbol{Z} \mid x \le z\}\\
&\mathtt{truncate}\, x \triangleq\\
&\quad \mathbf{if}\ 0 \le x\ \mathbf{then}\ \mathtt{floor}\, x\ \mathbf{else}\ \mathtt{ceiling}\, x
\end{aligned}
$$

これらは実数をそれに近い整数に変換する．`floor` は $-\infty$ に向かい、`ceiling` は $+\infty$ に向かい、`truncate` は 0 に向かって最も近い整数を対応させる．

条件式 $\mathbf{if}\ c\ \mathbf{then}\ x\ \mathbf{else}\ y$ は $c \in \boldsymbol{Bool}$ が **true** なら x、**false** なら y を表す．条件式は c について strict だが x, y については non strict とする．$\exists x\, P(x) \to \exists! x\, P(x)$ なる述語について、条件式 $\mathbf{if}\ \exists! x\, P(x)\ \mathbf{then}\ f\, x\ \mathbf{else}\ y$ は一意的に存在したら $f\, x$ さもなくば y を表す．またケース式 $\mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow v_1 \cdots\ l_n \Rightarrow v_n$ も使用する．これは $e = l_i$ ならば v_i を表す．ケース式は e, l_i について strict、v_i について non strict である．ケース式を使う場合は $\exists! i\ e = l_i$ でなければならない．

関数型の型式は右結合である．すなわち $\alpha \to \beta \to \gamma$ は $\alpha \to (\beta \to \gamma)$．関数適用 $f(x)$ は必要なければ括弧を略して $f\, x$ と書く．関数適用は左結合である．すなわち $f\, x\, y = (f\, x)\, y$．結合順位については、関数適用が最も結合度が高く、次に単項演算子 (形式上関数と同様引数の左に置かれるが、英数字以外の記号で構成されるもの)、最後に二項演算子の順である．すなわち $f\, x * -\, g\, x\, y = (f\, x) * (-((g\, x)\, y))$ 関数 f の定義域を $\mathtt{dom}\, f$ で表す．また関数 f の $X \subset \mathtt{dom}\, f$ への制限を $f|_X$ で表す．

$\lambda x.M$ はラムダ記法であり、$f(x) \triangleq M$ なる関数 f を表す．これはラムダカリキュラスのラムダ式ではない (本ドキュメントではラムダカリキュラスは使用しない)．これは主に演算子のシグネチャを与えるために使用されるので、簡潔さのために慣例に合わせて束縛変数は 1 文字であるとする．つまり $\lambda xy.M = \lambda x.(\lambda y.M)$．

関数引数は一般に括弧を略すが、意味記述において、意味関数定義の関数引数及び定義右辺に現れるシンタックスオブジェクトは $[\![\ ]\!]$ で囲む．これは意味関数ではシンタックスオブジェクトとセマンティックスオブジェクトが入り乱れるので、定義中のどこがシンタックスオブジェクトかを明示するためだけの慣習である．とくに本ドキュメントに限って言えば、シンタックスオブジェクトを S 式で表現している都合上、例えば $\mathcal{E}[\![(\mathtt{CONVSX}\ t\ x_1)]\!] \triangleq \cdots$ なる記述において、$[\![\ ]\!]$ がないと混乱の恐れがある．意味関数及びその補助関数以外では原則としてシンタックスオブジェクトでも $[\![\ ]\!]$ で囲まないが、使い分けは多分に気分の問題である．要は混乱をさけるためだけの記法であることを理解されたい．

ローカルな定義を行なうために「式 **where** 定義列」という表記を使用する．この場合定義列で定義される記号のスコープはこの表記内である．以下は例である．

$$
\begin{aligned}
&\mathtt{big} \triangleq f\, 100\\
&\qquad \mathbf{where}\\
&\qquad\quad f\ :\ \boldsymbol{N} \to \boldsymbol{N}\\
&\qquad\quad f\, 0 \triangleq 1\\
&\qquad\quad f\, n \triangleq n * f\,(n-1)\quad \mathbf{if}\ n \neq 0
\end{aligned}
$$

この f の定義は引数により場合分けをした定義の例にもなっている．通常数学で行なわれるこのような定義法も使用する．定義右辺にある「**if** 条件」は条件が満たされる場合のみその定義が使われることを意味する．

依存型は引数の「値」に依存して結果の型が決定する関数の型を表す場合に使用する．以下は依存型をもつ関数の定義例である．

$$
\begin{aligned}
&\mathtt{zeros}\ :\ n{:}\,\boldsymbol{N} \to \boldsymbol{N}^n\\
&\mathtt{zeros}\, n \triangleq (\underbrace{0, \ldots, 0}_{n\text{ 個}})
\end{aligned}
$$

X に $\bot_X$ を追加した集合を $X_\bot$ で表す．混乱がないならば $\bot_X$ は $\bot$ と略す．$X_\bot$ は flat domain $(X_\bot, \sqsubseteq)$ と考える．部分関数 $f{:}\,X \to Y$ は全関数と flat domain により $f{:}\,X \to Y_\bot$ という形で表す．すなわち $\forall x \in X\ x \in \mathtt{dom}\, f \leftrightarrow f\, x \neq \bot$ である．関

数空間のオーダー $\sqsubseteq$ は通常どうり $f \sqsubseteq g \stackrel{\triangle}{=} fx \in \mathtt{dom} f \rightarrow fx = gx$ で定義する．意味定義で現れる関数とコンストラクタは特に断らない限り全てストリクトであるとする．すなわち $f\colon X \to Y$ という関数定義には $f(\bot_X) \stackrel{\triangle}{=} \bot_Y$ という定義が付随しているとする．関数型 $\tau \stackrel{\triangle}{=} \alpha \to \beta_\bot$ について、$\bot_\tau$ は $\forall x \in \alpha \;\; fx = \bot_\beta$ なる関数 f とする．

リスト型 $[\tau]$ は後に説明する S 式のリストとは別ものである．要素は全て同一の型でなければならない．リスト型 $[\tau]$ のコンストラクタも [] で表す．すなわち $x_1, \ldots, x_n$ $(x_i \in \tau)$ のリストは $[x_1, \ldots, x_n] \in [\tau]$ で表す．リスト演算として以下を使用する．

$$
\begin{array}{l}
\#\colon [\tau] \to \boldsymbol{N} \\
\#[x_1, x_2, \ldots, x_n] \stackrel{\triangle}{=} n \\
\lambda x n.(x\,!\,n)\colon [\tau] \to \boldsymbol{N} \to \tau_\bot \\
x@[x_0, x_1, \ldots, x_n, \cdots]\,!\,n \stackrel{\triangle}{=} x_n \quad \textbf{if } n < \#x \\
x\,!\,n \stackrel{\triangle}{=} \bot \quad \textbf{if } n \geq \#x \\
\lambda e x.(e \in x)\colon \tau \to [\tau] \to \boldsymbol{Bool} \\
e \in [x_0, x_1, \ldots] \stackrel{\triangle}{=} \exists x_i \;\; e = x_i \\
\lambda x y.(x +\!\!+ y)\colon [\tau] \to [\tau] \to [\tau] \\
[x_1, \ldots, x_n] +\!\!+ [y_1, \ldots, y_m] \\
\quad \stackrel{\triangle}{=} [x_1, \ldots, x_n, y_1, \ldots, y_m] \\
\mathtt{reverse}\colon [\tau] \to [\tau] \\
\mathtt{reverse}\,[x_1, \ldots, x_n] \stackrel{\triangle}{=} [x_n, \ldots, x_1]
\end{array}
$$

$\#x$ でリスト x の長さを表す．定義の左辺にはこのようにパターンを使用する．定義左辺の引数パターンにおいて *italic* 体で書かている変数にマッチしたものが右辺で参照される．$x\,!\,i$ はリスト x の i 番目の要素を表す．$+\!+$ はリストの結合、reverse は反転である．ここで定義左辺 「x@ パターン」の x は定義の右辺で「パターン」がマッチしたもの全体を表す．

関数 $f\colon X \to Y_\bot$ は集合論における通常の解釈に従い直積 $X \times Y$ の部分集合と考える．空集合は従って関数型の $\bot$ である．ただしその直積の要素は分かりやすさのために $x \mapsto y$ で表す．例えば $f\,x_i \stackrel{\triangle}{=} y_i \;\; i \in I$(インデックス集合) なる部分関数 $f\colon X \to Y_\bot$ は $\{x_i \mapsto y_i \mid i \in I\}$ という集合で表すことも出来る． インデックス集合が明らかな場合は単に $\{x_i \mapsto y_i\}$ とも書く．

関数 $f\colon X \to Y_\bot$ について、$f := \;\; x_1 \mapsto y_1$ により、$gx = \textbf{if } x = x_1 \textbf{ then } y_1 \textbf{ else } fx$ なる関数 $g\colon X \cup \{x_1\} \to Y$ を表す．また $\sigma_i \stackrel{\triangle}{=} x_i \mapsto y_i$ のとき $f := [\sigma_1, \ldots, \sigma_n] \stackrel{\triangle}{=} ((f := \sigma_1) := \sigma_2) \cdots := \sigma_n$ とする．また関数 $f, g\colon X \to Y_\bot$ について、$f := g$ により、$hx = \textbf{if } gx \neq \bot \textbf{ then } gx \textbf{ else } fx$ なる関数 $h\colon X \to Y_\bot$ を表す．

レコード $r\colon \{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}$ について、$r := \{.l_{n_1} = v_{n_1}, \cdots, .l_{n_k} = v_{n_k}\}$ により、$r'.l_i = \textbf{if } i \in \{n_1, \ldots, n_k\} \textbf{ then } v_i \textbf{ else } r.l_i$ なる r' を表す．また関数の場合と同様に $r := [\sigma_1, \ldots, \sigma_n] \stackrel{\triangle}{=} ((r := \sigma_1) := \sigma_2) \cdots := \sigma_n$ とする．r_1, r_2 が同じ型のレコードのとき、$r_1 = r_2 := s$ なる s を $r_1 \div r_2$ で表す．関数及びレコードに対する $:=$ は両オペランドについて strict である．

A.1.1 シンタックスの記述

本ドキュメントではシンタックスは非終端記号を <> で囲むスタイルの拡張 BNF で与える． [] で囲むことでオプショナル、{} で囲むことでゼロ回以上の繰り返しを表すとする．また「< と空白」 及び「空白と >」で囲むことでグルーピングを表す．すなわち < *something* > を含んだ BNF 定義はそれらを新たな非終端記号 <*freshname*> で置き換え、定義 <*freshname*> ::= *something* を追加したものと同等である．BNF 定義において <> で囲まれた記号でそのシンタックスオブジェクトの集合を表す．

本ドキュメントでは構造をもつデータを記述するために S 式を使う．すなわちシンタックスオブジェクトの汎用的な媒体として S 式を使用するのである．以下の BNF で定義される Sexp が S 式の集合である．この BNF は S 式の具象構文を与えると同時に、上記の規約により S 式の集合 Sexp も定義している．

```
<Sexp>   ::= <Atom> | <List>
<List>   ::= ( {<Sexp>} )
<Atom>   ::= <Symbol> | <String>
           | <Fixnum> | <Flonum>
<Symbol> ::= シンボル
<String> ::= 文字列
<Fixnum> ::= 整数定数
<Flonum> ::= 浮動小数
```

本ドキュメントでは Sexp 上の関数として以下を定義する．

$\#\colon \texttt{List} \to \boldsymbol{N}$
$\#(x_1\ x_2\ \cdots\ x_n) \triangleq n$
$\lambda xn.(x\,!\,n)\colon \texttt{List} \to \boldsymbol{N} \to \texttt{Sexp}_\perp$
$x@(x_0\ x_1\ \cdots\ x_n\ \cdots)\,!\,n \triangleq x_n \quad \textbf{if } n < \#x$
$x\,!\,n \triangleq \perp \quad \textbf{if } n \geq \#x$
$\lambda ex.(e \in x)\colon \texttt{Sexp} \to \texttt{List} \to \boldsymbol{Bool}$
$e \in (x_0\ x_1\ \cdots) \triangleq \exists x_i\ \ e = x_i$
$\lambda xy.(x +\!+ y)\colon \texttt{List} \to \texttt{List} \to \texttt{List}$
$(x_1\ \cdots\ x_n) +\!+ (y_1\ \cdots\ y_m)$
$\quad \triangleq (x_1\ \cdots\ x_n\ y_1\ \cdots\ y_m)$

$\#\ e$ はリスト e の長さを表す．空リスト $()$ の長さは 0 である．$x\,!\,n$ は $x \in \texttt{List}$ の n 番目の要素、それがなければ $\perp$ を表す．$\in$ は集合論の記号だが、このようにリストにも流用する．また $[\tau]$ 型のリストと同様に $x +\!+ y$ で $x, y \in \texttt{List}$ の結合を表す．

シンタックスに関する記述の際、S 式のある部分集合 X の要素からなるリスト全体を正確に表すために以下の表記を導入する．

$$X\ \texttt{List} \triangleq \{(x_0\ x_1\ \cdots) \in \texttt{List} \mid x_i \in X\}$$

尚、S 式の集合としての X List と本ドキュメントで意味記述に使用する型 τ のリスト型 $[\tau]$ は別のものであり、混同しないように注意すること．紛らわしい場合には S 式の List あるいは $[\tau]$ 型のリストというように言葉を区別して使う．意味記述に使う型に S 式の部分集合を使うことは自由であり、$[X]$ という型も許される．しかし X List は S 式によるリストであり、$[X]$ とは別ものである．

A.2 LIR のシンタックス

ここでは LIR のシンタックスを二つに分けて定義する．図 **6** は L 式及びその部分的なシンタックスを定義する．図 **7** は L プログラムまでの上位の構造を定義する．

A.2.1 シンタックスに関する用語、記号の定義

L モジュール $(\texttt{MODULE}\ name\ alist\ \cdots) \in \texttt{Lmod}$ の $name$ をモジュール名、$alist$ を グローバル連想リストと呼ぶ．また、L 関数 $(\texttt{FUNCTION}\ name\ alist\ seq) \in \texttt{Lfunc}$ の $name$ を **L** 関数名 と呼び、$alist, seq$ をそれぞれ L 関数の ローカル連想リスト、L 式列という．これらについて以下の記法を定義する．

$\lambda m.(m.\texttt{name})\colon \texttt{Lmod} \to \texttt{String}$
$(\texttt{MODULE}\ name\ \cdots).\texttt{name} \triangleq name$
$\lambda m.(m.\texttt{alist})\colon \texttt{Lmod} \to \texttt{GlobalAlist}$
$(\texttt{MODULE}\ n\ alist\ \cdots).\texttt{alist} \triangleq alist$
$\lambda f.(f.\texttt{name})\colon \texttt{Lfunc} \to \texttt{String}$
$(\texttt{FUNCTION}\ name\ \cdots).\texttt{name} \triangleq name$
$\lambda f.(f.\texttt{alist})\colon \texttt{Lfunc} \to \texttt{LocalAlist}$
$(\texttt{FUNCTION}\ n\ alist\ \cdots).\texttt{alist} \triangleq alist$
$\lambda f.(f.\texttt{seq})\colon \texttt{Lfunc} \to Lseq$
$(\texttt{FUNCTION}\ n\ a\ seq\ \cdots).\texttt{seq} \triangleq seq$

L 連想リスト $(\texttt{ALIST}\ e_1\ e_2\ \cdots) \in \texttt{Lalist}$ の e_i を連想リストのエントリという．各エントリ e_i の第一要素は名前 (String) 第二要素はシンボルである．この名前をエントリ名、シンボルをエントリ e_i のエントリクラスという．

$$\begin{aligned} \lambda e.(e.\texttt{name}) &: \texttt{Ent} \to \texttt{String} \\ (name\ class\ \cdots).\texttt{name} &\triangleq name \\ \lambda e.(e.\texttt{class}) &: \texttt{Ent} \to \texttt{EntClass} \\ (name\ class\ \cdots).\texttt{class} &\triangleq class \end{aligned}$$

各エントリのエントリクラス以下のシンタックスはそのエントリクラスに依存する．以下の関係に注意 (直和ではない)．

$$\texttt{Lalist} = \texttt{GlobalAlist} \cup \texttt{LocalAlist}$$

集合 FooExp の要素を Foo 式と呼ぶ．例えば「r は Reg 式である」という表現は「$r \in \texttt{RegExp}$」と同じであり、「Reg 式 r について ...」は「$r \in \texttt{RegExp}$ について ...」と同じである．

式 $e \in \texttt{Lexp}$ はリストの形をしている．その第一要素は機能を定めるシンボルであり、**L** 式のコード という．L 式 e のコードを $e.\texttt{code}$ で表す．コードが FOO である式を FOO 式ともいう．Typed 式の場合は第二要素にその式の型を表す L タイプがくる．それを **L** 式の型といい、$e.\texttt{type}$ で表す．

$$\begin{aligned} \lambda e.(e.\texttt{code}) &: \texttt{Lexp} \to \texttt{Symbol} \\ (code\ \cdots).\texttt{code} &\triangleq code \\ \lambda e.(e.\texttt{type}) &: \texttt{TypedExp} \to \texttt{Ltype} \\ (code\ type\ \cdots).\texttt{type} &\triangleq type \end{aligned}$$

L 式 e のコードと型 (もしあれば) 以後に現れている L 式を出現順にならべたものを **L** 式の引数リストといい、$e.\texttt{args}$ で表す．正確な定義は以下のとおりである．

```
 1  <Lexp>              ::= <TypedExp> | <UnTypedExp>
 2  <TypedExp>          ::= <AtomicTypedExp> | <NonAtomicTypedExp>
 3  <AtomicTypedExp>    ::= <ConstExp> | <AddrExp> | <RegExp>
 4  <NonAtomicTypedExp> ::= <PureExp> | <MemExp> | <SetExp>
 5  <UnTypedExp>        ::= <JumpExp> | <DefLabelExp> | <CallExp> | <InterfaceExp>
 6                        | <SpecialExp>
 7  <Ltype>             ::= <Itype> | <Ftype> | <Fixnum>
 8  <Itype>             ::= I8 | I16 | I32 | I64 | I128
 9  <Ftype>             ::= F32 | F64 | F128
10  <ConstExp>          ::= <IntConstExp> | <FloatConstExp>
11  <IntConstExp>       ::= (INTCONST   <Ltype> <Fixnum>)
12  <FloatConstExp>     ::= (FLOATCONST <Ltype> <Flonum>)
13  <AddrExp>           ::= <StaticExp> | <FrameExp> | <LabelExp>
14  <StaticExp>         ::= (STATIC <Ltype> <String>)
15  <FrameExp>          ::= (FRAME  <Ltype> <String>)
16  <LabelExp>          ::= (LABEL  <Ltype> <String>)
17  <RegExp>            ::= <SimpleRegExp> | <SubRegExp>
18  <SimpleRegExp>      ::= (REG <Ltype> <String>)
19  <SubRegExp>         ::= (SUBREG <Ltype> <SimpleRegExp> <Fixnum> [& <Modifier>])
20  <PureExp>           ::= (<PureOps> <Ltype> <TypedExp> {<TypedExp>} [& <Modifier>])
21  <PureOps>           ::= <ArithOps> | <ConvOps> | <BitOps> | <ShiftOps> | <TstOps>
22                        | ASMCONST | PURE
23  <ArithOps>          ::= NEG | ADD | SUB | MUL | DIVS | DIVU | MODS | MODU
24  <ConvOps>           ::= CONVSX | CONVZX | CONVIT | CONVFX
25                        | CONVFT | CONVFI | CONVSF | CONVUF
26  <BitOps>            ::= BAND | BOR | BXOR | BNOT
27  <ShiftOps>          ::= LSHS | LSHU | RSHS | RSHU
28  <TstOps>            ::= TSTEQ | TSTNE | TSTLTS | TSTLES | TSTGTS | TSTGES
29                        | TSTLTU | TSTLEU | TSTGTU | TSTGEU
30  <MemExp>            ::= (MEM <Ltype> <TypedExp> [& <Modifier>])
31  <SetExp>            ::= (SET <Ltype> < <MemExp> | <RegExp> > <TypedExp>)
32  <JumpExp>           ::= (JUMP <LabelExp>)
33                        | (JUMPC <TypedExp> <LabelExp> <LabelExp>)
34                        | (JUMPN <TypedExp> ( { (<Fixnum> <LabelExp>) } ) <LabelExp>)
35  <DefLabelExp>       ::= (DEFLABEL <String>)
36  <CallExp>           ::= (CALL <TypedExp> ( {<TypedExp>} ) ( {<InterfaceVar>} ))
37  <InterfaceVar>      ::= <RegExp> | (MEM <Ltype> <FrameExp>)
38  <InterfaceExp>      ::= <PrologueExp> | <EpilogueExp>
39  <PrologueExp>       ::= (PROLOGUE (<Fixnum> <Fixnum>) {<InterfaceVar>})
40  <EpilogueExp>       ::= (EPILOGUE (<Fixnum> <Fixnum>) {<TypedExp>})
41  <SpecialExp>        ::= <ParallelExp> | <UseExp> | <ClobberExp>
42  <ParallelExp>       ::= (PARALLEL {<SetExp> | <CallExp> | <UseExp> | <ClobberExp>})
43  <UseExp>            ::= (USE <RegExp>)
44  <ClobberExp>        ::= (CLOBBER < <RegExp> | <MemExp> >)
45  <Modifier>          ::= <ShiftModifier> | <SubregModifier> | <MemModifier>
46  <ShiftModifier>     ::= (< S | U > <Fixnum> < D | U >)
47  <SubregModifier>    ::= S | N
48  <MemModifier>       ::= N | V
```

図 **6** L 式のシンタックス

```
 1  <Lprog>       ::= ( {<Lmod>} )
 2  <Lmod>        ::= (MODULE <String> <GlobalAlist> {<Ldata> | <Lfunc>})
 3  <GlobalAlist> ::= (ALIST {<GlobalEnt>} )
 4  <LocalAlist>  ::= (ALIST {<LocalEnt>} )
 5  <Lalist>      ::= <GlobalAlist> | <LocalAlist>
 6  <GlobalEnt>   ::= (<String> < <RegEnt> | <StaticEnt> >)
 7  <LocalEnt>    ::= (<String> < <RegEnt> | <FrameEnt>  >)
 8  <Ent>         ::= <GlobalEnt> | <LocalEnt>
 9  <RegEnt>      ::= REG    <Ltype> <Offset>
10  <FrameEnt>    ::= FRAME  <Ltype> <Align> <Offset>
11  <StaticEnt>   ::= STATIC < <Ltype> | UNKNOWN > <Align> <Segment> <Linkage>
12  <EntClass>    ::= REG | FRAME | STATIC
13  <Align>       ::= <Fixnum>
14  <Offset>      ::= <Fixnum>
15  <Segment>     ::= <String>
16  <Linkage>     ::= LDEF | XDEF | XREF
17  <Ldata>       ::= (DATA <String> {<DataSeq> | <ZeroSeq> | <SpaceSeq>})
18  <DataSeq>     ::= (<Ltype> {<Fixnum> | <Flonum> | <Lexp>})
19  <ZeroSeq>     ::= (ZEROS <Fixnum>)
20  <SpaceSeq>    ::= (SPACE <Fixnum>)
21  <Lfunc>       ::= (FUNCTION <String> <LocalAlist> <Lseq>)
22  <Lseq>        ::= ( {<Lexp>} )
```

図 7 L プログラムのシンタックス

$\lambda e.(e.\mathtt{args})\colon \mathtt{Lexp} \to \mathtt{Lexp\ List}$

$e.\mathtt{args} \triangleq ()$ **if** $e \in \mathtt{AtomicTypedExp}$

$e@(op\ type\ x_1\ \cdots\ x_n\ [\&\ m]).\mathtt{args}$
$\triangleq (x_1\ \cdots\ x_n)$ **if** $e \in \mathtt{NonAtomicTypedExp}$

$(\mathtt{JUMP}\ l_1).\mathtt{args} \triangleq (l_1)$

$(\mathtt{JUMPC}\ x\ l_1\ l_2).\mathtt{args} \triangleq (x\ l_1\ l_2)$

$(\mathtt{JUMPN}\ x\ ((c_1\ l_1)\cdots(c_n\ l_n))\ l_0).\mathtt{args}$
$\triangleq (x\ l_1\ \cdots\ l_n\ l_0)$

$(\mathtt{DEFLABEL}\ s).\mathtt{args} \triangleq ()$

$(\mathtt{CALL}\ x_0\ (x_1\ \cdots\ x_n)\ (y_1\ \cdots\ y_m)).\mathtt{args}$
$\triangleq (x_0\ x_1\ \cdots\ x_n\ y_1\ \cdots\ y_m)$

$(\mathtt{PROLOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n).\mathtt{args}$
$\triangleq (x_1\ \cdots\ x_n)$

$(\mathtt{EPILOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n).\mathtt{args}$
$\triangleq (x_1\ \cdots\ x_n)$

$(\mathtt{PARALLEL}\ x_1\ \cdots\ x_n).\mathtt{args} \triangleq (x_1\ \cdots\ x_n)$

$(\mathtt{USE}\ r).\mathtt{args} \triangleq (r)$

$(\mathtt{CLOBBER}\ x).\mathtt{args} \triangleq (x)$

L 式 e について $(a_1\ \cdots\ a_n) \triangleq e.\mathtt{args}$ とするとき、a_i を 直接の部分 **L** 式 あるいは第 i 引数 **L** 式という．上記定義の最初の二行が Typed 式に関するものである．AtomicTyped 式の引数リストは空である．それ以外の Typed 式は型に続く L 式でモディファイア (後述) を除く部分が引数リストとなる．これ以外の定義は UnTyped 式に関するものである．

x が e の直接の部分式である、という関係を $x \in_{\mathbf{a}} e$ で表す．またその推移的閉包を部分 **L** 式といい、$\in^{+}_{\mathbf{a}}$ で表す．同様にその反射的推移的閉包を広義の部分 **L** 式といい、$\in^{*}_{\mathbf{a}}$ で表す．

$x \in_{\mathbf{a}} e \triangleq x \in e.\mathtt{args}$

$x \in^{+}_{\mathbf{a}} e \triangleq x \in_{\mathbf{a}} e \vee \exists x_i \in e.\mathtt{args}\ x \in^{+}_{\mathbf{a}} x_i$

$x \in^{*}_{\mathbf{a}} e \triangleq x = e \vee x \in^{+}_{\mathbf{a}} e$

L 式列 $es \triangleq (e_0\ e_1\ \cdots)$ において e_i を es のトップレベル **L** 式という．$es!i = e$ のとき、es のロケーション i の L 式が e であるという．

A.2.2 シンタックスレベルでの制約

ここでは BNF で表現出来ていないシンタックス上の制約を定義する．セマンティックスが定義可能なシンタックスオブジェクト X を正しい X という．例えば正しい L 式、正しい L プログラム、などと表現する．以下はその判定関数である．

$$
\begin{aligned}
\texttt{validprog} &: \texttt{Lprog} \to \boldsymbol{Bool}\\
\texttt{validmod} &: \texttt{Lmod} \to \boldsymbol{Bool}\\
\texttt{validalist} &: \texttt{Lalist} \to \boldsymbol{Bool}\\
\texttt{validdata} &: \texttt{Ldata} \to \boldsymbol{Bool}\\
\texttt{validfunc} &: \texttt{Lfunc} \to \boldsymbol{Bool}\\
\texttt{validseq} &: \texttt{Lseq} \to \boldsymbol{Bool}\\
\texttt{validexp} &: \texttt{Lexp} \to \boldsymbol{Bool}\\
\texttt{validltype} &: \texttt{Ltype} \to \boldsymbol{Bool}
\end{aligned}
$$

以下は validltype, validexp, validseq の定義である．補助関数として validexp_set, validexp_sem を使っており、前者は Set 式の出現についての制約、後者は各 L 式に依存する制約を表す．

$$
\begin{aligned}
&\texttt{validltype}\, t \triangleq\\
&\quad t \in \boldsymbol{Z} \to 0 < t \wedge t \equiv 0 \pmod 8\\
&\texttt{validexp}\, e \triangleq\\
&\quad \texttt{validexp\_set}\, e \wedge \forall x \in^{*}_{\mathtt{a}} e\ \ \texttt{validexp\_sem}\, x\\
&\texttt{validexp\_set}\, e \triangleq\\
&\quad \textbf{if}\ e \in \texttt{ParallelExp}\\
&\quad \textbf{then}\ \forall x \in_{\mathtt{a}} e\ \ \texttt{validexp}\, x\\
&\quad \textbf{else}\ \forall x \in^{+}_{\mathtt{a}} e\ \ x \notin \texttt{SetExp}\\
&\texttt{validexp\_sem} e \triangleq \text{後に定義される}\\
&\texttt{validseq}\, s \triangleq\\
&\quad \wedge\ \ \forall e \in s\ \ \texttt{validexp}\, e\\
&\quad \wedge\ \ \forall e \in s\ \ e \in \texttt{UnTypedExp} + \texttt{SetExp}\\
&\quad \wedge\ \ \texttt{mklabelenv}\, s \neq \texttt{DUP}\\
&\quad \wedge\ \ \forall e \in s\ \ (\exists x \in^{+}_{\mathtt{a}} e\ \ x \in \texttt{LabelExp})\\
&\qquad\quad \to e \in \texttt{JumpExp}\\
&\quad \wedge\ \ \forall e \in s\ \ (\forall x \in^{+}_{\mathtt{a}} e\ \ x \in \texttt{LabelExp}\\
&\qquad\quad \to (\texttt{mklabelenv}\, s)\,(x\,!\,2) \neq \bot)
\end{aligned}
$$

ここに mklabelenv は L 式列を引数とし、ラベル文字列にそれが定義 (DEFLABEL) されているロケーションを対応させる部分関数 (後に定義される L 環境の LabelEnv) を返す関数である．ただし二重定義が存在したらシンボル DUP を返す．この関数の「つまらない定義」が後の L 環境の定義の所で与えられる．

すなわち正しい L タイプは、それが Fixnum であれば正の値でなければならない．正しい L 式は Parallel 式を除き、Set 式を部分式として含んではならない．また正しい L 式列は正しい L 式のリストで、トップレベル式が UnTyped 式と Set 式のみからなるものであり、ラベル式は Jump 式のオペランド以外には現れず、Label 式に対応するラベルは一意的に DefLabel 式で定義されていなければならない．validexp_sem は後に L 式のセマンティックスの所で定義される．

A.3 LIR のセマンティックス

以下が中間言語 LIR の意味関数一覧である．これらを（数学的な）関数として書くことにより LIR のセマンティックスを与える．

$$
\begin{aligned}
\mathcal{P} &: \texttt{Lprog} \to \text{L プログラムの意味}\\
\mathcal{M} &: \texttt{Lmod} \to \text{L モジュールの意味}\\
\mathcal{A} &: \texttt{Lalist} \to \text{L 連想リストの意味}\\
\mathcal{D} &: \texttt{Ldata} \to \text{L データの意味}\\
\mathcal{F} &: \texttt{Lfunc} \to \text{L 関数の意味}\\
\mathcal{S} &: \texttt{Lseq} \to \text{L 式列のステップ実行の意味}\\
\mathcal{E} &: \texttt{Lexp} \to \text{L 式の意味}\\
\mathcal{T} &: \texttt{Ltype} \to \text{L タイプの意味}
\end{aligned}
$$

L 式列はいわばコードセグメントに置かれたマシン命令列を意味している．各 L 式 e_i は順時実行されることでメモリに対して読み書きを行なう．メモリは L 式列の一部でない．したがって別に提供する必要がある．これを **L** メモリと呼ぶ．また一部の L 式はレジスタやアドレスを表しているので、それらの対応も与える必要がある．これを **L** 環境と呼ぶ．

ここではまず L メモリと L 環境の直観的意味を説明し、それを定式化する．そしての実行環境における L 式の動作の意味を与えることにより意味関数を定義する．

A.3.1 L メモリと L 環境の直観的意味

L メモリは以下のものからなる．それぞれに関連する L 式の直観的な意味も説明する．ここで $A \in \texttt{Itype}$ はアドレスを表す型である．

(1) プログラムカウンター pc.
L 関数の L 式列 $(e_0\, e_1\, \cdots)$ で次に実行すべき命令のロケーションを示す．この pc は LIR のレジスタでは表現できない．$\texttt{pc} = i$ のとき e_i の実行後、pc は通常 1 増えるが、Jump 命令の場合は Label 式の表すロケーションが設定される．

(2) プログラムメモリ pm.
L 関数が置かれているメモリ．命令がメモリにどのように配置されるかはマシンに依存しすぎるし、その表現を必要とする最適化は特殊なものに限られる．よって我々はプログラムメモリが保持する対象を個々の命令の具体的ビットパターンではなく、シンタックスオブジェクトとしての L 関数とする．本来プログラムメモリはセマンティックスの世界に属するものであるが、中間言語ではプログラム変換が重要であるから、プログラムメモリが保持するものはシン

タックスオブジェクトとしている．このメモリにアクセス出来るのは唯一 Call 式のみであり、それも L 関数のセマンティックスを通じてのアクセスに制限する．このようにモデル化しても、上でいう特殊な最適化をこの枠組で行なうことは可能である．詳細はプラグマティックスで説明する．

(3) レジスタメモリ `rm`.
汎用レジスタ群が置かれているメモリ．意味定義の都合によりレジスタは次のデータメモリとはアドレス空間の異なるレジスタメモリに置かれているとする．よってレジスタのアドレスが存在する．しかしそれはデータメモリのアドレス空間とは異なり、データメモリにアクセスする命令によってレジスタをアクセスすることは出来ない．
Reg 式 (`REG` t s) は $s \in$ `String` で指定される型 t のレジスタが格納している値を表す．また Set 式 (`SET` t (`REG` t s) v) は型 t の Typed 式 v の値をそのレジスタの新しい値とするように書き込みを行なう．特に `"@FP"`という文字列で表されるレジスタをフレームポインタという．

(4) データメモリ `dm`.
データの置かれている読み書き可能なメモリ．型 A の任意の Typed 式は `dm` のアドレスを表すと解釈出来る．Static 式 (`STATIC` A s) は $s \in$ `String` で指定される `dm` のアドレスを表す．Frame 式 (`FRAME` A s) は $s \in$ `String` で示されるオフセット値を現在のフレームポインタの値に加えて生成される `dm` のアドレスを表す．このとき s をフレーム変数と呼ぶ．
Mem 式 (`MEM` t a) は `dm` のアドレス a にある型 t の オブジェクトを意味し、それが格納している値を表す．また Set 式 (`SET` t (`MEM` t a) v) は型 t の Typed 式 v の値をそのオブジェクトの新しい値とするように書き込みを行なう．

(5) トレース `tr`.
メモリへの参照により IO を行なうマシンでは関数の意味として特定のメモリへの読み書きを時系列として記録したリストを考慮する必要がある．そのリストを保持するための変数である．このような変数は実際のハードウエアには存在しない．これはあくまで意味記述の便宜上の変数である．読み書きがトレースへ記録されるオブジェクトを **volatile** オブジェクト という．volatile オブジェクトの意味定義には次のランダムステートも関連する．

(6) ランダムステート `rs`.
意味記述で使われる乱数発生関数 `random` の状態 (実数) を保持する変数．ここを参照するのはこの `random` 関数のみである．乱数は意味記述において、ある特定の値を予測することで可能となるような (あやまった) 最適化を禁止するために使われる．例えば volatile 変数 `x` について `x-x` を 0 とするのは誤りである．我々は volatile 変数に、その値は常に乱数であるという意味づけをすることで、この問題に対処している．

L 環境は以下のものからなる．

(1) Reg 式の対応 `reg`.
関数中の全ての Reg 式 (`REG` t s) の $s \in$ `String` に `rm` 中の適当なアドレス `reg` s を対応させる．これにより (`REG` t s) は `rm` 中のそのアドレスに置かれた型 t のレジスタを表すことになる．

(2) Static 式の対応 `sta`.
関数中の全ての Static 式 (`STATIC` A s) の $s \in$ `String` に `dm` 中の適当なアドレス `sta` s を対応させる．これにより (`STATIC` A s) は `dm` 中のそのアドレスを表すことになる．

(3) Frame 式の対応 `fra`.
関数中の全ての Frame 式 (`FRAME` A s) の $s \in$ `String` にフレームポインタからの適当なオフセット値 `fra` s を対応させる．これにより (`FRAME` A s) は現在のフレームポインタ値とそのオフセット値の和で作られる `dm` 中のアドレスを表すことになる．

(4) Label 式の対応 `lab`.
関数中の全ての Label 式 (`LABEL` A s) の $s \in$ `String` に関数中のロケーション `lab` s を対応させる．これにより (`LABEL` A s) は関数中のそのロケーションを表すことになる．

以上のような L メモリと L 環境が具体的に一つあたえられたとすると L 式そして L 関数の意味が確定する．L 式列 $s \in$ `Lseq` の 1 ステップ実行の意味を $\mathcal{S}[\![s]\!]$ と書くとすると、それは以下のような型をもつ関数である．

$\mathcal{S}[\![s]\!]$: 環境 $\rightarrow$ メモリ $\rightarrow$ メモリ

すなわち $\mathcal{S}[\![s]\!]$ とは、環境を引数とし、直前のメモリから、s の 1 ステップの実行後のメモリを返すステートトランスフォーマを返す関数である．

以上の考え方で LIR のセマンティックスを定義していく．意味定義に使われる定数、関数、あるいは型

が以下のように定義されている場合.

$$foo \triangleq \text{Unspecified}$$

これは foo の具体的な定義を LIR のドキュメントとしては与えない、という意味である. 具体的な定義が個々のマシンにより異なるというケースで、一般的には定義することが不可能な場合、定義をこのように Unspecified としておく. ただし、マシンにより異なるとはいっても、意味定義の都合上必要な条件が foo に課される場合があるが、それは個別に記述される. このように定義が Unspecified と扱われている定数を以後便宜上 Unspecified 定数と呼ぶ. 関数、型についても同様である. これらの具体的な定義はマシン記述の情報から定義される. マシン記述の情報と Unspecified 関数との関係はマシン記述に関するドキュメントで定義される.

A.3.2 ビット列とバイトの定義

データを表すドメインとして、ビット列を定義する. また、メモリでアドレス指定される最小単位のビット列としてバイトを定義する. 現役のハードウエアを考慮し 1 バイトは 8 ビットと定義する.

$$\begin{aligned}
\mathtt{Bit} &: \boldsymbol{N} \to \boldsymbol{Type} \\
\mathtt{Bit}\, n &\triangleq \{0,1\}^n \\
\mathtt{Bits} &: \boldsymbol{Type} \\
\mathtt{Bits} &\triangleq \textstyle\bigcup_{n=0}^{\infty} \mathtt{Bit}\, n \\
\# &: \mathtt{Bits} \to \boldsymbol{N} \\
\#\, b_{n-1}\cdots b_0 &\triangleq n \\
\# &: \{\mathtt{Bit}\, n \mid n \in \boldsymbol{N}\} \to \boldsymbol{N} \\
\#\, \mathtt{Bit}\, n &\triangleq n \\
\mathtt{Byte} &: \boldsymbol{Type} \\
\mathtt{Byte} &\triangleq \mathtt{Bit}\, 8
\end{aligned}$$

$\mathtt{Bit}\, n$ は $\{0,1\}$ の n 直積であり n ビットのデータからなる型を表す. Bits は全ての $n \in \boldsymbol{N}$ についての $\mathtt{Bit}\, n$ の和集合であり、意味領域として使用する. $\#b$ で b のビット数を表す. また n ビットのビット列の集合 $\mathtt{Bit}\, n$ に対しても $\#\,\mathtt{Bit}\, n$ は n を表すとする. ビット列 $b \in \mathtt{Bit}\, n$ 上の操作記述では、その各ビットを並べ $b = b_{n-1}\cdots b_0$ と表記することで各ビットを参照する. $\mathtt{Bit}\, 0$ は集合論的に 1 点集合 $\{\emptyset\}$ となる. $\emptyset \in \mathtt{Bits}$ は実際は何も値を返さないが、形式上 Bits の値を返すとすると意味定義で扱いやすいような関数が返す値として使用される.

以下はビット列を数学的な数と解釈する関数である. $\boldsymbol{N}$ は理想的な符号なし整数、$\boldsymbol{Z}$ は理想的な符号つき整数、$\boldsymbol{R}$ は理想的な浮動小数と考えている.

$$\begin{aligned}
\mathtt{nb} &: \mathtt{Bits} \to \boldsymbol{N} \\
\mathtt{nb}\ b_{n-1}\cdots b_0 &\triangleq \textstyle\sum_{i=0}^{n-1} b_i * 2^i \\
\mathtt{zb} &: \mathtt{Bits} \to \boldsymbol{Z} \\
\mathtt{zb}\ b_{n-1}\cdots b_0 &\triangleq \mathtt{nb}\ b_{n-1}\cdots b_0 - b_{n-1} * 2^n \\
\mathtt{rb} &: \mathtt{Bits} \to \boldsymbol{R}_\perp \\
\mathtt{rb}\, b &\triangleq b \text{ の IEEE 浮動小数解釈}
\end{aligned}$$

$\mathtt{rb}\, b$ で $b \in \mathtt{Bits}$ が IEEE 解釈で実数とならない場合 $\mathtt{rb}\, b = \perp$ である. 以下は数学的な数をビット列に変換する関数である. br も rb 同様部分関数であり、$\mathtt{br}\, n\, x$ で x が IEEE 解釈の n ビット表現を持たない場合は $\mathtt{rb}\, n\, x = \perp$ である.

$$\begin{aligned}
\mathtt{bz} &: n{:}\boldsymbol{N} \to \boldsymbol{Z} \to \mathtt{Bit}\, n \\
\mathtt{bz}\, n\, x &\triangleq !b \in \mathtt{Bit}\, n \;\; x \equiv \mathtt{nb}\, b \pmod{2^n} \\
\mathtt{br} &: n{:}\boldsymbol{N} \to \boldsymbol{R} \to \mathtt{Bit}\, n_\perp \\
\mathtt{br}\, n\, x &\triangleq x \text{ の } n \text{ ビット IEEE 浮動小数解釈}
\end{aligned}$$

A.3.3 L タイプのセマンティックス ($\mathcal{T}$)

$\mathtt{Ltype} = \mathtt{Itype} + \mathtt{Ftype} + \mathtt{Fixnum}$ である. 型 t が I または F で始まる十進数のとき、その値を t のビット幅とよぶ. Fixnum の場合はそれがビット幅である. 制約 validltype によって、それは 8 で割り切れる正の整数である.

$\mathcal{T}$ は L タイプの意味関数であり、与えられた L タイプの意味としての型を返す. 直観的には $\mathcal{T}[\![\mathtt{I}n]\!]$ は n ビットの整数型、$\mathcal{T}[\![\mathtt{F}n]\!]$ は n ビットの浮動小数型を表すと考えたいかも知れない. しかし、我々の定義では以下に示すように $\mathtt{I}n$ と $\mathtt{F}n$ に同一のセマンティックスを与えている.

$$\begin{aligned}
\mathcal{T} &: \mathtt{Ltype} \to \boldsymbol{Type} \\
\mathcal{T}[\![\mathtt{I}n]\!] &\triangleq \mathtt{Bit}\, n \\
\mathcal{T}[\![\mathtt{F}n]\!] &\triangleq \mathtt{Bit}\, n \\
\mathcal{T}[\![n]\!] &\triangleq \mathtt{Bit}\, n \\
\mathtt{Bitw} &: \boldsymbol{Type} \\
\mathtt{Bitw} &\triangleq \{n \in \boldsymbol{N} \mid \exists t \in \mathtt{Ltype}\ \mathcal{T}[\![t]\!] = \mathtt{Bit}\, n\}
\end{aligned}$$

符号つきであるか符号なしであるか、あるいは浮動小数であるか、という区別はタグつきアーキテクチャのようなものを除き、COINS で対象としているハードではデータそのものの型ではなく本来は演算で区別されるものである. セマンティックスの定義でもそれを反映し、単にそのビット幅 n を持つ $\mathtt{Bit}\, n$ 型と扱う. Bitw は意味記述においてビット幅の型であることを明示するために使われる.

A.3.4 L メモリの定義

L メモリは関数カウンタ、レジスタメモリ、データ

メモリからなる．バイト単位でアドレスが付いているものとする．すなわちメモリ m とは $\boldsymbol{N}$ から Byte への部分関数 m: $\boldsymbol{N} \to \mathtt{Byte}_\perp$ と考える．よって $\mathtt{dom}\, m$ が実際のアドレス空間を表す．このアドレス空間は後に説明するようにフレームのアロケーションにより拡張される．

以下はロケーション、プログラムメモリのアドレス、レジスタメモリのアドレス、そしてデータメモリのアドレスの型定義である．実際には全て $\boldsymbol{N}$ であり、この定義は単にドキュメント上の便宜である．

$$\begin{aligned}\mathtt{Location} &\triangleq \boldsymbol{N}\\ \mathtt{PMemAddr} &\triangleq \boldsymbol{N}\\ \mathtt{RMemAddr} &\triangleq \boldsymbol{N}\\ \mathtt{DMemAddr} &\triangleq \boldsymbol{N}\end{aligned}$$

以下はプログラムメモリの型 PMem、レジスタメモリの型 RMem、及びデータメモリの型 DMem の定義である．

$$\begin{aligned}\mathtt{PMem}, \mathtt{RMem}, \mathtt{DMem} &: \boldsymbol{Type}\\ \mathtt{PMem} &\triangleq \mathtt{PMemAddr} \to \mathtt{Lfunc}_\perp\\ \mathtt{RMem} &\triangleq \mathtt{RMemAddr} \to \mathtt{Byte}_\perp\\ \mathtt{DMem} &\triangleq \mathtt{DMemAddr} \to \mathtt{Byte}_\perp\end{aligned}$$

意味記述におけるメモリの扱いで、やや技巧的なものがあるので、ここで説明しておく．意味記述ではメモリが部分関数として表現されていることを利用して L 関数の実行中にのみ存在するフレームを表現する．たとえば m:DMem というメモリに L 関数の実行により生まれたフレームメモリ m':DMem (ここに $\mathtt{dom}\, m' \cap \mathtt{dom}\, m = \emptyset$) を追加したものは $m'' \triangleq m := m'$ と表される．L 関数の実行においてはフレームもそれ以外の部分も修正されるが、L 関数終了時にはフレームが解放され、もとの m のうちフレーム以外の部分の修正のみが L 関数の副作用として記録されなければならない．それはすなわち m'' のドメインを元の $\mathtt{dom}\, m$ に制限したものに他ならない．つまり L 関数の副作用は $m''|_{\mathtt{dom}\, m}$ と表現出来る．同様のテクニックはレジスタメモリにも適用されている．詳細は L 関数のセマンティックスを参照のこと．

メモリへの参照により IO を行なうようなマシンでは関数の意味に特定のメモリへの書き込み、読み込みを時系列として記録したリストを含める必要がある．このような外部とのインタラクションを表現するために、アクションの時系列情報を記録するトレースというリストが L メモリには存在する．

$$\begin{aligned}\mathtt{Action}, \mathtt{Trace} &: \boldsymbol{Type}\\ \mathtt{Action} &\triangleq \mathtt{Symbol} \times \mathtt{Ltype} \times [\mathtt{Bits}]\\ \mathtt{Trace} &\triangleq [\mathtt{Action}]\end{aligned}$$

アクションのリスト型が Trace である．以下は現在定義されているアクション、そのアクションを直接記録する (可能性のある) L 式のコード、そのアクションの表現である．

処理	Action
読み込み	(READ, $ltype$, [$address$])
書き込み	(WRITE, $ltype$, [$address, value$])

以上をまとめたものが L メモリとなる．以下の Mem が L メモリの型である．最後の rs はランダムステートである．

$$\begin{aligned}\mathtt{Mem} &: \boldsymbol{Type}\\ \mathtt{Mem} &\triangleq \{\, \mathtt{pc}\colon \mathtt{Location},\\ &\qquad \mathtt{pm}\colon \mathtt{PMem},\\ &\qquad \mathtt{rm}\colon \mathtt{RMem},\\ &\qquad \mathtt{dm}\colon \mathtt{DMem},\\ &\qquad \mathtt{tr}\colon \mathtt{Trace},\\ &\qquad \mathtt{rs}\colon \boldsymbol{R}\,\}\end{aligned}$$

意味記述の簡潔さのために、アクションをトレースに追加する関数を定義する．

$$\begin{aligned}&\mathtt{addtotr}\\ &\quad : \mathtt{Mem} \to \mathtt{Symbol} \to \mathtt{Ltype} \to [\mathtt{Bits}] \to \mathtt{Mem}\\ &\mathtt{addtotr}\, \sigma\, s\, [\![t]\!]\, b\\ &\quad \triangleq \sigma := \{\sigma.\mathtt{tr} = \sigma.\mathtt{tr} +\!+ [(s, [\![t]\!], b)]\}\end{aligned}$$

メモリの Endianness や境界条件はマシンに依存するので、メモリへの読み書き関数を Unspecified な部分関数とせざるおえない．具体的な定義では以下のメモリ関係の補助関数を利用する．

$$\texttt{bits2bytes}\colon \texttt{Bit}(n * \#\texttt{Byte}) \to [\texttt{Byte}]$$
$$\texttt{bits2bytes}\, w_{n-1} \cdots w_1 w_0 \triangleq [w_0, w_1, \ldots, w_{n-1}]$$
$$\texttt{bytes2bits}\colon l\colon [\texttt{Byte}] \to \texttt{Bit}(\#l * \#\texttt{Byte})$$
$$\texttt{bytes2bits}\, [w_0, w_1, \ldots, w_{n-1}] \triangleq w_{n-1} \cdots w_1 w_0$$
$$\texttt{readbytes} : (\boldsymbol{N} \to \texttt{Byte}) \to \boldsymbol{N} \to \boldsymbol{N} \to [\texttt{Byte}]$$
$$\texttt{readbytes}\ m\, l\, n \triangleq [ml, m(l+1), \ldots, m(l+n-1)]$$
$$\texttt{writebytes} : (\boldsymbol{N} \to \texttt{Byte}) \to \boldsymbol{N} \to \boldsymbol{N} \to [\texttt{Byte}] \to (\boldsymbol{N} \to \texttt{Byte})$$
$$\texttt{writebytes}\ m\, l\, n\, [b_0, b_1, \ldots, b_{n-1}] \triangleq m := [l \mapsto b_0, l+1 \mapsto b_1, \ldots, l+n-1 \mapsto b_{n-1}]$$

bits2bytes と bytes2bits はワードと Byte のリストの相互変換である．readbytes $m\,l\,n$ はメモリ m のアドレス l から n バイトをリストの形で読む．writebytes $m\,l\,b$ はメモリ m のアドレス l からバイトのリスト b で与えられたバイトを順時書き込む．(ただし数学的な関数として定義するために、破壊的に書き込むのではなく、対応を修正したメモリを返す、という形で定義している．)

以下はメモリやレジスタアクセスに関する L 式の意味定義で使われる関数である．これらはトレースにはアクセスしない．

$$\texttt{rmread}\colon \texttt{Mem} \to \texttt{RMemAddr} \to t\colon \texttt{Ltype} \to \mathcal{T}[\![t]\!]_\perp$$
$$\texttt{rmread} \triangleq \text{Unspecified}$$
$$\texttt{dmread}\colon \texttt{Mem} \to \texttt{DMemAddr} \to t\colon \texttt{Ltype} \to \mathcal{T}[\![t]\!]_\perp$$
$$\texttt{dmread} \triangleq \text{Unspecified}$$
$$\texttt{rmwrite} : \texttt{Mem} \to \texttt{RMemAddr} \to t\colon \texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Mem}_\perp$$
$$\texttt{rmwrite} \triangleq \text{Unspecified}$$
$$\texttt{dmwrite} : \texttt{Mem} \to \texttt{DMemAddr} \to t\colon \texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Mem}_\perp$$
$$\texttt{dmwrite} \triangleq \text{Unspecified}$$

rmread $\sigma\, a\, [\![t]\!]$ はレジスタメモリ σ.rm のアドレス a にある型 t のオブジェクトの値を型 $\mathcal{T}[\![t]\!]$ の値として返す．dmread もデータメモリに対する同様の関数である．rmwrite $\sigma\, a\, [\![t]\!]\, w$ はレジスタメモリ σ.rm のアドレス a に、型 t の値 w を書き込む (もちろん正確に言えば σ は破壊しないで、修正された新たなメモリを返す)．dmwrite も同様である．dmread と dmwrite は部分関数である．これにより有限なメモリ、バウンダリ条件を表現する．

メモリ関連の定義の最後に「乱数発生関数」を定義する．乱数は意味記述において、なにか不定な値（⊥ではない）が代入されるということを表すため、及びある特定の値を予測することで可能となるような（あやまった）最適化を禁止するために使われる．

乱数発生関数 random を数学的な関数として定義するために、L メモリのランダムステートの値から何らかの方法で「乱数」と「ランダムステートの値を次の状態に変更した L メモリ」を返すものとする．この関数で作られた乱数を予測不能値という．意味記述の便宜のために random 自体は以下のように引数として L メモリを取るとし、本質的な乱数関数は rand とする．そして rand は Unspecified 関数とする．その意図はそのインプリメントを最適化ルーチンに予測出来ないようにすることである．

$$\texttt{random} : t\colon \texttt{Ltype} \to \texttt{Mem} \to \texttt{Mem} \times \mathcal{T}[\![t]\!]$$
$$\texttt{random}[\![t]\!]\, \sigma \triangleq (\sigma := \{\texttt{rs} = r\}, v) \ \textbf{where}\ (r, v) \triangleq \texttt{rand}[\![t]\!]\, \sigma.\texttt{rs}$$
$$\texttt{rand} : t\colon \texttt{Ltype} \to \boldsymbol{R} \to \boldsymbol{R} \times \mathcal{T}[\![t]\!]$$
$$\texttt{rand} \triangleq \text{Unspecified}$$

A.3.5 L 環境の定義

L 環境は L 式と Mem の対応である．Frame 式の引数文字列が対応するのはオフセット値であり、その値とフレームポインタの値の和が Frame 式の表すものであった．フレームの伸びる向きはインプリメント依存なので、オフセットの型は $\boldsymbol{Z}$ とする．

$$\texttt{FrameOffset} : \boldsymbol{Type}$$
$$\texttt{FrameOffset} \triangleq \boldsymbol{Z}$$

以下は L 環境を構成するテーブルの型の定義である．

$$\texttt{RegEnv} \triangleq \texttt{String} \to \texttt{RMemAddr}_\perp$$
$$\texttt{StaticEnv} \triangleq \texttt{String} \to \texttt{DMemAddr}_\perp$$
$$\texttt{FrameEnv} \triangleq \texttt{String} \to \texttt{FrameOffset}_\perp$$
$$\texttt{LabelEnv} \triangleq \texttt{String} \to \texttt{Location}_\perp$$

このように定義上は全てのテーブルを全関数として定義する．すなわち L 関数に現れてもいない全てのシンボルや文字列について何らかの割り当てを行なっているものをテーブルとする．これらテーブルを集めたものが、L 環境となる．以下の Env が L 環境の型

である.

$$\begin{aligned}&\texttt{Env} \;:\; \boldsymbol{Type}\\ &\texttt{Env} \triangleq \{\, \texttt{reg}\!:\texttt{RegEnv},\\ &\qquad\quad \texttt{sta}\!:\texttt{StaticEnv},\\ &\qquad\quad \texttt{fra}\!:\texttt{FrameEnv},\\ &\qquad\quad \texttt{lab}\!:\texttt{LabelEnv}\,\}\end{aligned}$$

以下は L 式列からラベル環境を作る mklabelenv の定義である. L 式列中の DefLabel 式の文字列にそのロケーションを対応させる部分関数を返す. ただし二重定義が存在したらシンボル DUP を返す. 正しい L 式列ではこのようなことは起きない（実は validseq の定義にこの関数を使っている）. ラベル環境については L 式列から直ちに定まる. にもかかわらず LIR の表現ではラベル文字列を介した表現にした理由は人間にとっての見やすさを考慮してのことであり、インプリメントではラベルの参照を直接データ構造のリンクで表現することも考えられる.

$$\begin{aligned}&\texttt{mklabelenv} \;:\; \texttt{Lseq} \to \texttt{LabelEnv} + \{\texttt{DUP}\}\\ &\texttt{mklabelenv}\, s \triangleq \text{略}\end{aligned}$$

A.3.6 L 式のセマンティックス $(\mathcal{S}, \mathcal{E})$

$\mathcal{S}$ が L 式列 s の 1 ステップ実行の意味関数である. $\mathcal{S}[\![s]\!]$ へ具体的な環境 ρ:Env を引数として与えると 1 ステップ実行を行なう Mem 間のステートトランスフォーマを返す.

$$\begin{aligned}&\mathcal{S} \;:\; \texttt{Lseq} \to \texttt{Env} \to \texttt{Mem} \to \texttt{Mem}_\perp\\ &\mathcal{S}[\![s]\!]\rho\,\sigma \triangleq \sigma_2\\ &\qquad\textbf{where}\\ &\qquad\sigma_1 \triangleq \sigma :=\{.\texttt{pc} = \sigma.\texttt{pc}+1\}\\ &\qquad(\sigma_2, v) \triangleq \mathcal{E}[\![s\,!\,\sigma.\texttt{pc}]\!]\rho\,\sigma_1\end{aligned}$$

ここで $\mathcal{E}$ はひとつの L 式の意味関数であり、変更された Mem と式の値の組を返す.

$$\mathcal{E} : \texttt{Lexp} \to \texttt{Env} \to \texttt{Mem} \to (\texttt{Mem} \times \texttt{Bits})_\perp$$

$\mathcal{E}$ は UnTypedExp にも適用される. その場合、値は $\emptyset$ となる. 以下、各 L 式に対してその意味記述を行なう. また同時に validexp_sem も定義される. 以下は意味記述の例である:

$(\texttt{CONVSX}\; t\; x_1)$ $\quad t, t_1 \in \texttt{Itype} \;\wedge\; w > w_1$
符号拡張.

$$\mathcal{E}[\![(\texttt{CONVSX}\; t\; x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\, w\, (\texttt{zb}\, v_1))$$

各意味記述において、簡単のために次の規約を置く. 記述に現れる x_i が $x_i \in \texttt{TypedExp}$ で、記号 $t_i, w, w_i, \sigma_i, v_i$ がその記述では定義されていない場合、以下の定義がその記述内で使用出来るとする. 尚、この定義は副作用のある引数が通常は出現順に左から右へと評価されることを意味している.

$$\begin{aligned}t_i &\triangleq x_i.\texttt{type}\\ w &\triangleq \#\mathcal{T}[\![t]\!]\\ w_i &\triangleq \#\mathcal{T}[\![t_i]\!]\\ (\sigma_1, v_1) &\triangleq \mathcal{E}[\![x_1]\!]\rho\sigma\\ (\sigma_2, v_2) &\triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1\\ &\;\;\vdots\\ (\sigma_i, v_i) &\triangleq \mathcal{E}[\![x_i]\!]\rho\sigma_{i-1}\end{aligned}$$

上記意味記述の左上の L 式を e とする. 実際は e のさらに左端に参照番号が現れるが、それは意味記述の部分ではない. まず記述内に定義の無い $t_1, w, w_1, \sigma_1, v_1$ が先の規約に従い定義される. e に続く制約条件が validexp_sem e となる. この場合は以下の定義がこの意味記述に付随しているとする.

$$\begin{aligned}&\texttt{validexp\_sem}\,[\![(\texttt{CONVSX}\; t\; x_1)]\!]\\ &\quad\triangleq t, t_1 \in \texttt{Itype} \;\wedge\; w > w_1\end{aligned}$$

制約条件がない場合は validexp_sem $e \triangleq$ **true** である. その次に言葉による説明と、$\mathcal{E}[\![e]\!]$ のフォーマルな定義が与えられる. ある意味記述内で、その意味定義に必要な Unspecified 関数を定義することがある. また複数の意味記述で共有できるような補助関数を定義することもある. これらの関数のスコープは本ドキュメント全体である. 意味関数は L 式のパターンごとに分けて定義されているが、どのパターンにもマッチしない e については validexp_sem $e \triangleq$ **false** とする.

A.3.6.1 Const 式

ここでは Const 式に対する $\mathcal{E}$ を定義する. ASMCONST 式は Pure 式に分類されている.

(1) $(\texttt{INTCONST}\; t\; z)$ $\quad t \in \texttt{Itype}$
$z \in \boldsymbol{Z}$ を型 t で表した整数値を表す.

$$\mathcal{E}[\![(\texttt{INTCONST}\; t\; z)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{bz}\, w\, z)$$

(2) $(\texttt{FLOATCONST}\; t\; r)$ $\quad t \in \texttt{Ftype} \wedge \texttt{br}\, w\, r \neq \perp$
$r \in \boldsymbol{R}$ を型 t で表した浮動小数値を表す.

$$\mathcal{E}[\![(\texttt{FLOATCONST}\; t\; r)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{br}\, w\, r)$$

いくつかの意味記述では記述内で Const 式を作り、記述に使用しているので、Const 式を作る関数をここで

定義しておく.

$$\texttt{mkconst}: t{:}\,\texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Lexp}$$
$$\texttt{mkconst}[\![t]\!]\,b \triangleq \textbf{if } t \in \texttt{Itype} \textbf{ then } [\![(\texttt{INTCONST}\ t\ z)]\!] \textbf{ where } z \triangleq \texttt{zb}\,b \textbf{ else } [\![(\texttt{FLOATCONST}\ t\ r)]\!] \textbf{ where } r \triangleq \texttt{rb}\,b$$

A.3.6.2 Addr 式

ここでは Addr 式に対する $\mathcal{E}$ を定義する. Static 式はグローバル変数のアドレス、Frame 式はローカル変数のアドレスを表すためのものである. Label 式は L 関数のロケーションを表す.

(1) $(\texttt{STATIC}\ t\ s)$ $t \in \texttt{Itype}$
$s \in \texttt{String}$ で示されるプログラムメモリまたはデータメモリのアドレス.

$$\mathcal{E}[\![(\texttt{STATIC}\ t\ s)]\!]\rho\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\rho.\texttt{sta}[\![s]\!]))$$

(2) $(\texttt{FRAME}\ t\ s)$ $t \in \texttt{Itype}$
$s \in \texttt{String}$ で示されるオフセットとフレームポインタの和で生成されるデータメモリのアドレス.

$$\mathcal{E}[\![(\texttt{FRAME}\ t\ s)]\!]\rho\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\texttt{nb}(\texttt{rmread}\,\sigma(\rho.\texttt{reg}[\![\texttt{"@FP"}]\!])[\![t]\!]) + \rho.\texttt{fra}[\![s]\!]))$$

(3) $(\texttt{LABEL}\ t\ s)$ $t \in \texttt{Itype}$
$s \in \texttt{String}$ で示されるロケーション.

$$\mathcal{E}[\![(\texttt{LABEL}\ t\ s)]\!]\rho\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\rho.\texttt{lab}[\![s]\!]))$$

A.3.6.3 Reg 式

ここでは Reg 式に対する $\mathcal{E}$ を定義する. マシンの自然なワード長より小さな単位での演算等で、オペランドに指定されたレジスタの一部にアクセスする場合がある. そのような部分的なレジスタを、元のレジスタの部分レジスタという.

以下の定義が使われるのは Reg 式が Set 式の第一引数以外で、そのレジスタが格納している値を取り出すという場合にかぎられるように Set 式が定義されている.

(1) $(\texttt{REG}\ t\ s)$
$s \in \texttt{String}$ で示されるレジスタの値.

$$\mathcal{E}[\![(\texttt{REG}\ t\ s)]\!]\rho\sigma \triangleq (\sigma, \texttt{rmread}\,\sigma\,(\rho.\texttt{reg}[\![s]\!])\,[\![t]\!])$$

(2) $(\texttt{SUBREG}\ t\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m]\ )$ $w < w_1 \wedge 0 \le n < w_1/w \in \boldsymbol{N}$
第一引数で指定されるレジスタの n 番目の部分レジスタの値. $m \in \texttt{SubregModifier}$ はここでは参照されない. このモディファイアの意味については Set 式の意味記述を参照のこと.

$$\mathcal{E}[\![(\texttt{SUBREG}\ t\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m])]\!]\rho\sigma \triangleq (\sigma, \texttt{rmread}\,\sigma(\rho.\texttt{reg}[\![s]\!] + \texttt{subregoffset}[\![t]\!]\,[\![t_1]\!]\,n)\,[\![t]\!])$$

ここに $\texttt{subregoffset}$ は型 t_1 のレジスタ中で n 番目の型 t の部分レジスタのオフセットを与える Unspecified 関数である.

$$\texttt{subregoffset} : \texttt{Ltype} \to \texttt{Ltype} \to \boldsymbol{N} \to \boldsymbol{N}$$
$$\texttt{subregoffset} \triangleq \text{Unspecified}$$

A.3.6.4 Pure 式

ここでは Pure 式に対する $\mathcal{E}$ を定義する.

(1) $(\texttt{NEG}\ t\ x_1\ )$ $t = t_1$
符号反転.

$$\mathcal{E}[\![(\texttt{NEG}\ t\ x_1)]\!]\rho\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(-\texttt{zb}\,v_1)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{NEG}\ t\ x_1)]\!]\rho\sigma \triangleq (\sigma_1, \texttt{br}\,w\,(-\texttt{rb}\,v_1)) \quad \textbf{if } t \in \texttt{Ftype}$$

(2) $(\texttt{ADD}\ t\ x_1\ x_2\ )$ $t = t_1 = t_2$
加算.

$$\mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 + \texttt{zb}\,v_2)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 + \texttt{rb}\,v_2)) \quad \textbf{if } t \in \texttt{Ftype}$$

(3) $(\texttt{SUB}\ t\ x_1\ x_2\ )$ $t = t_1 = t_2$
減算.

$$\mathcal{E}[\![(\texttt{SUB}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 - \texttt{zb}\,v_2)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{SUB}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 - \texttt{rb}\,v_2)) \quad \textbf{if } t \in \texttt{Ftype}$$

(4) $(\texttt{MUL}\ t\ x_1\ x_2\ )$ $t = t_1 = t_2$
乗算.

$$\mathcal{E}[\![(\texttt{MUL}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 * \texttt{zb}\,v_2)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{MUL}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 * \texttt{rb}\,v_2)) \quad \textbf{if } t \in \texttt{Ftype}$$

(5) (DIVS $t\ x_1\ x_2$) $t = t_1 = t_2$
符号つき除算.

$$\mathcal{E}[\![(\texttt{DIVS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divz}\,w\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2))\ \textbf{if}\ t \in \texttt{Itype}$$

$$\mathcal{E}[\![(\texttt{DIVS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divr}\,w\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2))\ If\ t \in \texttt{Ftype}$$

$\texttt{divz}: w{:}\,\texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\,w_\perp$

$\texttt{divz}\,w\,n\,d \triangleq$
$\quad \textbf{if}\ d = 0\ \textbf{then}\ \perp\ \textbf{else}\ \texttt{bz}\,w\,(\texttt{truncate}(n/d))$

$\texttt{divr}: w{:}\,\texttt{Bitw} \to \boldsymbol{R} \to \boldsymbol{R} \to \texttt{Bit}\,w_\perp$

$\texttt{divr}\,w\,n\,d$
$\quad \triangleq\ \textbf{if}\ d = 0\ \textbf{then}\ \perp\ \textbf{else}\ \texttt{br}\,w\,(n/d)$

(6) (DIVU $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
符号なし除算.

$$\mathcal{E}[\![(\texttt{DIVU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divz}\,w\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

(7) (MODS $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
符号つき剰余.

$$\mathcal{E}[\![(\texttt{MODS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{modz}\,w\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2))$$

$\texttt{modz}: w{:}\,\texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\,w_\perp$

$\texttt{modz}\,w\,n\,d$
$\quad \triangleq\ \textbf{if}\ d = 0\textbf{then}\ \perp$
$\qquad \textbf{else}\ \texttt{bz}\,w\,(n - d * \texttt{truncate}(n/d))$

(8) (MODU $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
符号なし剰余.

$$\mathcal{E}[\![(\texttt{MODU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{modz}\,w\,\sigma_2\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

(9) (CONVSX $t\ x_1$) $t, t_1 \in \texttt{Itype} \wedge w > w_1$
符号拡張.

$$\mathcal{E}[\![(\texttt{CONVSX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{zb}\,v_1))$$

(10) (CONVZX $t\ x_1$) $t, t_1 \in \texttt{Itype} \wedge w > w_1$
ゼロ拡張.

$$\mathcal{E}[\![(\texttt{CONVZX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{nb}\,v_1))$$

(11) (CONVIT $t\ x_1$) $t, t_1 \in \texttt{Itype} \wedge w < w_1$
精度の低い整数へ.

$$\mathcal{E}[\![(\texttt{CONVIT}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{zb}\,v_1))$$

(12) (CONVFX $t\ x_1$) $t, t_1 \in \texttt{Ftype} \wedge w > w_1$
精度の高い浮動小数へ.

$$\mathcal{E}[\![(\texttt{CONVFX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{br}\,w\,(\texttt{rb}\,v_1))$$

(13) (CONVFT $t\ x_1$) $t, t_1 \in \texttt{Ftype} \wedge w < w_1$
精度の低い浮動小数へ.

$$\mathcal{E}[\![(\texttt{CONVFT}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{br}\,w\,(\texttt{rb}\,v_1))$$

(14) (CONVFI $t\ x_1$) $t \in \texttt{Itype} \wedge t_1 \in \texttt{Ftype}$
浮動小数から整数へ.

$$\mathcal{E}[\![(\texttt{CONVFI}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{truncate}(\texttt{rb}\,v_1)))$$

(15) (CONVSF $t\ x_1$) $t \in \texttt{Ftype} \wedge t_1 \in \texttt{Itype}$
符号つき整数から浮動小数へ.

$$\mathcal{E}[\![(\texttt{CONVSF}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{br}\,w\,(\texttt{zb}\,v_1))$$

(16) (CONVUF $t\ x_1$) $t \in \texttt{Ftype} \wedge t_1 \in \texttt{Itype}$
符号なし整数から浮動小数へ.

$$\mathcal{E}[\![(\texttt{CONVUF}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{br}\,w\,(\texttt{nb}\,v_1))$$

(17) (BAND $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
論理積.

$$\mathcal{E}[\![(\texttt{BAND}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{bitop}\,(\lambda xy.\ \textbf{if}\ x = 1 \wedge y = 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\,v_1\,v_2)$$

$\texttt{bitop}$
$\quad : (\texttt{Bit}\,1 \to \texttt{Bit}\,1 \to \texttt{Bit}\,1)$
$\qquad\qquad \to \texttt{Bit}\,n \to \texttt{Bit}\,n \to \texttt{Bit}\,n$

$\texttt{bitop}\,f\ x_{n-1} \cdots x_0\ y_{n-1} \cdots y_0$
$\quad \triangleq\ z_{n-1} \cdots z_0 \quad \textbf{where}\ z_i \triangleq f\,x_i\,y_i$

(18) (BOR $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
論理和.

$$\mathcal{E}[\![(\texttt{BOR}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{bitop}\,(\lambda xy.\ \textbf{if}\ x = 1 \vee y = 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\,v_1\,v_2)$$

(19) (BXOR $t\ x_1\ x_2$) $t \in \texttt{Itype} \wedge t = t_1 = t_2$
排他的論理和.

$$\mathcal{E}[\![(\texttt{BXOR}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{bitop}\,(\lambda xy.\ \textbf{if}\ x = y\ \textbf{then}\ 0\ \textbf{else}\ 1)\,v_1\,v_2)$$

(20) (BNOT $t\ x_1$) $t \in \texttt{Itype} \wedge t = t_1$
論理否定.

$$\mathcal{E}[\![(\texttt{BNOT}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bitop}\,(\lambda xy.\ \textbf{if}\ x = 1\ \textbf{then}\ 0\ \textbf{else}\ 1)\,v_1\,(\texttt{bz}\,w\,0))$$

(21) (LSHS $t\ x_1\ x_2$ [& m]) $t, t_2 \in \texttt{Itype} \wedge t = t_1$

四つのシフト命令において、左シフト、右シフトの区別はシフトカウントが正の場合の向きによる．符号つきシフトか符号なしシフトかの区別はシフトされる x_1 の値 v_1 の解釈による．

$$\mathcal{E}[\![(\mathtt{LSHS}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{genericshift}\ w\ (\mathtt{zb}\ v_1)\ (\mathtt{shiftcount}\ v_2\ [\![m]\!]))$$

$$\mathtt{genericshift}\colon\ w\colon \mathtt{Bit}w \to \boldsymbol{Z} \to \boldsymbol{Z} \to \mathtt{Bit}\,w$$

$$\mathtt{genericshift}\,w\,n\,c \triangleq \mathtt{bz}\,w\,(\mathtt{floor}(n * 2^c))$$

$$\mathtt{shiftcount}\colon \mathtt{Bits} \to \mathtt{ShiftModifier} \to \boldsymbol{Z}_\perp$$

$$\mathtt{shiftcount}\,b\,[\![(s\,n\,d)]\!] \triangleq \textbf{if } c_0 = c_1 \textbf{ then } c_0 \textbf{ else } \textbf{case } [\![d]\!] \textbf{ of } \mathtt{D} \Rightarrow c_1\ \mathtt{U} \Rightarrow \perp \textbf{ where } f \triangleq \textbf{case } [\![s]\!] \textbf{ of } \mathtt{S} \Rightarrow \mathtt{zb}\ \mathtt{U} \Rightarrow \mathtt{nb},\ c_0 \triangleq f\,b,\ c_1 \triangleq f(\mathtt{bz}\,n\,c_0)$$

(22) $(\mathtt{LSHU}\ t\ x_1\ x_2\ [\&\ m]\ )$ $t, t_2 \in \mathtt{Itype} \wedge t = t_1$
符号なし左シフト．

$$\mathcal{E}[\![(\mathtt{LSHU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\mathtt{LSHU}\ t\ x_1\ x_2\ \&\ (\mathtt{U}\ 5\ \mathtt{D}))]\!]$$

$$\mathcal{E}[\![(\mathtt{LSHU}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{genericshift}\ w\ (\mathtt{nb}\ v_1)\ (\mathtt{shiftcount}\ v_2\ [\![m]\!]))$$

(23) $(\mathtt{RSHS}\ t\ x_1\ x_2\ [\&\ m]\ )$ $t, t_2 \in \mathtt{Itype} \wedge t = t_1$
符号つき右シフト．

$$\mathcal{E}[\![(\mathtt{RSHS}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\mathtt{RSHS}\ t\ x_1\ x_2\ \&\ (\mathtt{U}\ 5\ \mathtt{D}))]\!]$$

$$\mathcal{E}[\![(\mathtt{RSHS}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{genericshift}\ w\ (\mathtt{zb}\ v_1)\ (-\mathtt{shiftcount}\ v_2\ [\![m]\!]))$$

(24) $(\mathtt{RSHU}\ t\ x_1\ x_2\ [\&\ m]\ )$ $t, t_2 \in \mathtt{Itype} \wedge t = t_1$
符号なし右シフト．

$$\mathcal{E}[\![(\mathtt{RSHU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\mathtt{RSHU}\ t\ x_1\ x_2\ \&\ (\mathtt{U}\ 5\ \mathtt{D}))]\!]$$

$$\mathcal{E}[\![(\mathtt{RSHU}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{genericshift}\ w\ (\mathtt{nb}\ v_1)\ (-\mathtt{shiftcount}\ v_2\ [\![m]\!]))$$

(25) $(\mathtt{TSTEQ}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
比較 $(x_1 = x_2)$.

$$\mathcal{E}[\![(\mathtt{TSTEQ}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x = y)\ (\mathtt{zb}\ v_1)\ (\mathtt{zb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Itype}$$

$$\mathcal{E}[\![(\mathtt{TSTEQ}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x = y)\ (\mathtt{rb}\ v_1)\ (\mathtt{rb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Ftype}$$

$$\mathtt{tstr}\colon w\colon \mathtt{Bit}w \to (\boldsymbol{R} \to \boldsymbol{R} \to \boldsymbol{Bool}) \to \boldsymbol{R} \to \boldsymbol{R} \to \mathtt{Bit}\,w$$

$$\mathtt{tstr}\,w\,f\,x\,y \triangleq \mathtt{bz}\,w\,\textbf{if } f\,x\,y \textbf{ then } 1 \textbf{ else } 0$$

(26) $(\mathtt{TSTNE}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
比較 $(x_1 \neq x_2)$.

$$\mathcal{E}[\![(\mathtt{TSTNE}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x \neq y)\ (\mathtt{zb}\ v_1)\ (\mathtt{zb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Itype}$$

$$\mathcal{E}[\![(\mathtt{TSTNE}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x \neq y)\ (\mathtt{rb}\ v_1)\ (\mathtt{rb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Ftype}$$

(27) $(\mathtt{TSTLTS}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
符号つき比較 $(x_1 < x_2)$.

$$\mathcal{E}[\![(\mathtt{TSTLTS}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x < y)\ (\mathtt{zb}\ v_1)\ (\mathtt{zb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Itype}$$

$$\mathcal{E}[\![(\mathtt{TSTLTS}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x < y)\ (\mathtt{rb}\ v_1)\ (\mathtt{rb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Ftype}$$

(28) $(\mathtt{TSTLES}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
符号つき比較 $(x_1 \leq x_2)$.

$$\mathcal{E}[\![(\mathtt{TSTLES}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x \leq y)\ (\mathtt{zb}\ v_1)\ (\mathtt{zb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Itype}$$

$$\mathcal{E}[\![(\mathtt{TSTLES}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \mathtt{tstr}\ w\ (\lambda xy.x \leq y)\ (\mathtt{rb}\ v_1)\ (\mathtt{rb}\ v_2)) \textbf{ if } t_1 \in \mathtt{Ftype}$$

(29) $(\mathtt{TSTGTS}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
符号つき比較 $(x_1 > x_2)$.

$$\mathcal{E}[\![(\mathtt{TSTGTS}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\mathtt{TSTLTS}\ t\ x_2\ x_1)]\!]$$

(30) $(\mathtt{TSTGES}\ t\ x_1\ x_2\ )$ $t \in \mathtt{Itype} \wedge t_1 = t_2$
符号つき比較 $(x_1 \geq x_2)$.

$$\mathcal{E}[\![(\texttt{TSTGES}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLES}\ t\ x_2\ x_1)]\!]$$

(31) $(\texttt{TSTLTU}\ t\ x_1\ x_2)$ $t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$
符号なし比較 $(x_1 < x_2)$.

$$\mathcal{E}[\![(\texttt{TSTLTU}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{tstr}\, w\, (\lambda xy.x < y)\, (\texttt{nb}\, v_1)\, (\texttt{nb}\, v_2))$$

(32) $(\texttt{TSTLEU}\ t\ x_1\ x_2)$ $t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$
符号なし比較 $(x_1 \leq x_2)$.

$$\mathcal{E}[\![(\texttt{TSTLEU}\ t\ x_1\ x_2)]\!]\rho\sigma \triangleq (\sigma_2, \texttt{tstr}\, w\, (\lambda xy.x \leq y)\, (\texttt{nb}\, v_1)\, (\texttt{nb}\, v_2))$$

(33) $(\texttt{TSTGTU}\ t\ x_1\ x_2)$ $t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$
符号なし比較 $(x_1 > x_2)$.

$$\mathcal{E}[\![(\texttt{TSTGTU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLTU}\ t\ x_2\ x_1)]\!]$$

(34) $(\texttt{TSTGEU}\ t\ x_1\ x_2)$ $t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$
符号なし比較 $(x_1 \geq x_2)$.

$$\mathcal{E}[\![(\texttt{TSTGEU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLEU}\ t\ x_2\ x_1)]\!]$$

(35) $(\texttt{ASMCONST}\ t\ x_1)$ $t = t_1 \wedge \texttt{isasmconst}\, x_1$
意味は x_1 と同じである.

$$\mathcal{E}[\![(\texttt{ASMCONST}\ t\ x_1)]\!] \triangleq \mathcal{E}[\![x_1]\!]$$
$$\texttt{isasmconst} : \texttt{Lexp} \to \boldsymbol{Bool}$$
$$\texttt{isasmconst} \triangleq \text{Unspecified}$$

ここに isasmconst はリンカが解決出来る定数かどうかを判定する Unspecified 関数であり、以下の条件を満たさなければならない.

$$\forall e \in \texttt{Lexp}\quad \texttt{isasmconst}\, e \to \forall x \in^*_{\texttt{a}} e$$
$$x \in \texttt{ConstExp} + \texttt{StaticExp} + \texttt{PureExp}$$

(36) $(\texttt{PURE}\ t\ x_1\ \cdots\ x_n)$ $x_1 \in \texttt{IntConstExp}$
副作用のない任意の演算. 拡張用.

$$\mathcal{E}[\![(\texttt{PURE}\ t\ x_1\ \cdots\ x_n)]\!]\rho\sigma \triangleq (\sigma_n, \texttt{pureapply}[\![t]\!]\, (\texttt{nb}\, v_1)\, [v_2, \ldots, v_n])$$
$$\texttt{pureapply}: t: \texttt{Ltype} \to \boldsymbol{N} \to [\texttt{Bits}] \to \mathcal{T}[\![t]\!]$$
$$\texttt{pureapply} \triangleq \text{Unspecified}$$

ここに pureapply は自然数 $\texttt{nb}\, v_1$ で示される関数に引数 $[v_2, \ldots, v_n]$ を与えた結果を計算する Unspecified 関数である.

A.3.6.5 Mem 式

ここでは Mem 式に対する $\mathcal{E}$ を定義する.

(1) $(\texttt{MEM}\ t\ x_1\ [\&\ m])$ $t_1 \in \texttt{Itype}$
データメモリのアドレス x_1 にある型 t のオブジェクトが格納している値. $m \in \texttt{MemModifier}$ が存在し、$m = \texttt{V}$ の時にかぎり Mem 式は volatile オブジェクトを表し、読み込みがトレースに記録され、結果は予測不能値となる.

$$\mathcal{E}[\![(\texttt{MEM}\ t\ x_1)]\!] \triangleq \mathcal{E}[\![(\texttt{MEM}\ t\ x_1\ \&\ \texttt{N})]\!]$$
$$\mathcal{E}[\![(\texttt{MEM}\ t\ x_1\ \&\ m)]\!]\rho\sigma \triangleq$$
$$\textbf{case}\ [\![m]\!]\ \textbf{of}$$
$$\texttt{N} \Rightarrow (\sigma_1, \texttt{dmread}\, \sigma_1\, (\texttt{nb}\, v_1)\, [\![t]\!])$$
$$\texttt{V} \Rightarrow \texttt{random}[\![t]\!](\texttt{addtotr}\, \sigma_1\, \texttt{READ}\, [\![t]\!]\, [v_1])$$

A.3.6.6 Set 式

ここでは Set 式に対する $\mathcal{E}$ を定義する.

(1) $(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ [\&\ m])\ x_2)$ $t = t' = t_2$
データメモリのアドレス x_1 にある型 t のオブジェクトへ x_2 の値を代入する. $m \in \texttt{MemModifier}$ が存在し、$m = \texttt{V}$ の時にかぎり Mem 式は volatile オブジェクトを表し、書き込みがトレースに記録される.

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1)\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ \&\ \texttt{N})\ x_2)]\!]$$
$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ \&\ m)\ x_2)]\!]\rho\sigma \triangleq (\sigma_4, v_2)$$
$$\textbf{where}$$
$$\sigma_3 \triangleq \texttt{dmwrite}\, \sigma_2 (\texttt{nb}\, v_1)\, [\![t]\!]\, v_2$$
$$\sigma_4 \triangleq \textbf{case}\ [\![m]\!]\ \textbf{of}$$
$$\texttt{N} \Rightarrow \sigma_3$$
$$\texttt{V} \Rightarrow \texttt{addtotr}\, \sigma_3\, \texttt{WRITE}\, [\![t]\!]\, [v_1, v_2]$$

(2) $(\texttt{SET}\ t\ (\texttt{REG}\ t'\ s)\ x_1)$ $t = t' = t_1$
Reg 式が表すレジスタに x_1 の値を代入する.

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{REG}\ t'\ s)\ x_1)]\!]\rho\sigma \triangleq (\texttt{rmwrite}\, \sigma_1\, (\rho.\texttt{reg}[\![s]\!])\, [\![t]\!]\, v_1, v_1)$$

(3) $(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m])\ x_1)$
$t = t' = t_1$
SubReg 式が表す部分レジスタに x_1 の値を代入する. $m \in \texttt{SubregModifier}$ が存在し、$m = \texttt{N}$ の時にかぎり $(\texttt{REG}\ t_1\ s)$ 中の指定部分以外に予測不能値が入る.

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n)\ x_1)]\!] \triangleq \mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ \&\ \texttt{S})\ x_1)]\!]$$

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ \&\ m)\ x_1)]\!]\rho\sigma \triangleq (\sigma_4, v_1)$$

where

$$a \triangleq \rho.\texttt{reg}[\![s]\!] + \texttt{subregoffset}[\![t]\!][\![t_1]\!]\,n$$

$$\sigma_3 \triangleq \textbf{case}\ [\![m]\!]\ \textbf{of}$$

$$\texttt{S} \Rightarrow \sigma_1$$

$$\texttt{N} \Rightarrow \texttt{rmwrite}\ \sigma_2\ (\rho.\texttt{reg}[\![s]\!])\ [\![t]\!]\ r$$

where

$$(\sigma_2, r) \triangleq \texttt{random}[\![t]\!]\,\sigma_1$$

$$\sigma_4 \triangleq \texttt{rmwrite}\,\sigma_3\,a\,[\![t]\!]\,v_1$$

A.3.6.7 Jump 式

ここでは Jump 式に対する $\mathcal{E}$ を定義する．簡潔さのため、ここの意味記述において以下を定義する．

$$j_i \triangleq \mathcal{E}[\![l_i]\!]\rho\sigma$$

(1) (JUMP l_1)
ラベル式 l_1 で示されるロケーションへジャンプする．

$$\mathcal{E}[\![(\texttt{JUMP}\ l_1)]\!]\rho\,\sigma \triangleq (\texttt{jmp}\,\sigma\,j_1, \emptyset)$$

$$\texttt{jmp} : \texttt{Mem} \to \texttt{Bits} \to \texttt{Mem}$$

$$\texttt{jmp}\,\sigma\,j \triangleq \sigma := \{\texttt{pc} = \texttt{nb}\,j\}$$

(2) (JUMPC $x_1\ l_1\ l_2$) $x_1.\texttt{code} \in \texttt{TstOps}$
Tst 式 x_1 の値が 1 なら l_1 へ、0 なら l_2 へジャンプする．

$$\mathcal{E}[\![(\texttt{JUMPC}\ x_1\ l_1\ l_2)]\!]\rho\sigma \triangleq (\texttt{jmp}\,\sigma_1\ \textbf{case}\ \texttt{nb}\,v_1\ \textbf{of}\ 1 \Rightarrow j_1\ 0 \Rightarrow j_2, \emptyset)$$

(3) (JUMPN $x_1\ ((c_1\ l_1)\cdots(c_n\ l_n))\ l_0$) $t_1 \in \texttt{Itype} \wedge i \neq j \to c_i \neq c_j$
整数式 x_1 の値が $c_i \in \boldsymbol{Z}$ に等しければ l_i へジャンプ、そうでなければ l_0 へジャンプする．

$$\mathcal{E}[\![(\texttt{JUMPN}\ x_1\ ((c_1\ l_1)\cdots(c_n\ l_n))\ l_0)]\!]\rho\,\sigma \triangleq (\texttt{jmp}\,\sigma_1\ \textbf{if}\ \exists!i\ c_i = \texttt{nb}\,v_1\ \textbf{then}\ j_i\ \textbf{else}\ j_0, \emptyset)$$

A.3.6.8 DefLabel 式

ここでは DefLabel 式に対する $\mathcal{E}$ を定義する．DefLabel 式は L 環境の LabelEnv を作るために mklabelenv で参照されるが、$\mathcal{E}$ では単に無視される．

(1) (DEFLABEL s)

$$\mathcal{E}[\![(\texttt{DEFLABEL}\ s)]\!]\rho\,\sigma \triangleq (\sigma, \emptyset)$$

A.3.6.9 Call 式

ここでは Call 式に対する $\mathcal{E}$ を定義する．

(1) (CALL $x_1\ (x_2\ \cdots\ x_n)\ (y_1\ \cdots\ y_m)$) $t_1 \in$ Itype
x_1 の値 v_1 で示されるプログラムメモリに L 関数 $\sigma.\texttt{pm}\,v_1 (\neq \bot_{\texttt{Lfunc}})$ があると仮定し、多値関数としての解釈 $\mathcal{F}[\![\sigma.\texttt{pm}\,v_1]\!]$ に引数を与え、結果の多値を y_i に代入する．

$$\mathcal{E}[\![(\texttt{CALL}\ x_1\ (x_2\ \cdots\ x_n)\ (y_1\ \cdots\ y_m))]\!]\rho\,\sigma \triangleq (\sigma'_m, \emptyset)$$

where

$$(\sigma'_0, [b_1, \ldots, b_m]) \triangleq \mathcal{F}[\![\sigma.\texttt{pm}\,v_1]\!]\,\rho\,[v_2, \ldots, v_n]\,\sigma_n$$

$$t_i \triangleq y_i.\texttt{type}$$

$$b'_i \triangleq \texttt{mkconst}[\![t_i]\!]\,b_i$$

$$(\sigma'_i, u_i) \triangleq \mathcal{E}[\![(\texttt{SET}\ t_i\ y_i\ b'_i)]\!]\,\rho\,\sigma'_{i-1}$$

A.3.6.10 Interface 式

Interface 式は基本的にフレームポインタの増減操作であるが、一般的な形での定義はまだ出来ていない．

A.3.6.11 Special 式

ここでは Special 式に対する $\mathcal{E}$ を定義する．以下の定義では副作用 x_i と x_j $(i < j$ とする$)$ が競合している場合、すなわち同一のオブジェクトを変更しようとしている場合、後の副作用 x_j が有効であるということも表している．

(1) (PARALLEL $x_1\ \cdots\ x_n$)
$x_1, \ldots, x_n$ の副作用を同時に行なう．

$$\mathcal{E}[\![(\texttt{PARALLEL}\ x_1\ \cdots\ x_n)]\!]\rho\,\sigma \triangleq (\sigma := [\sigma_1 \div \sigma, \ldots, \sigma_n \div \sigma], \emptyset)$$

where

$$(\sigma_i, v_i) \triangleq \mathcal{E}[\![x_i]\!]\rho\,\sigma$$

(2) (USE r)
レジスタ r が使用されることを表す．現在表示的な意味は与えられていない．

(3) (CLOBBER x_1)
構文から x_1 は Set 式の第一引数に許されるものであり、そこに予測不能値が入ることを表す．

$$\mathcal{E}[\![(\texttt{CLOBBER}\ x_1)]\!]\rho\,\sigma \triangleq \mathcal{E}[\![(\texttt{SET}\ t_1\ x_1\ c)]\!]\rho\,\sigma_2$$

where

$$(\sigma_2, r) \triangleq \texttt{random}[\![t_1]\!]\,\sigma_1$$

$$c \triangleq \texttt{mkconst}[\![t_1]\!]\,r$$

A.3.7 L 連想リストのセマンティックス $(\mathcal{A})$

L 連想リスト $a \in \texttt{Lalist}$ のセマンティックス $\mathcal{A}[\![a]\!]$ は引数のエントリクラス $(c{:}\,\texttt{EntClass})$ に依存した型

(EntType c) のレコードで、それは文字列からそのいろいろな属性を返す関数のレコードである．validalist の制約から L 連想リストのエントリは重複したエントリ名を持たないと仮定している．本付録では定義は省略する．

A.3.8 L 関数のセマンティックス ($\mathcal{F}$)

ここでは意味関数 $\mathcal{F}$ を定義する．ここに Args, Rets はそれぞれ引数と返却値の型である．L 関数は多値関数なので、返却値も Bits のリストになっている．

$$\begin{aligned}\mathcal{F} &: \mathtt{Lfunc} \to \mathtt{Env} \to \mathtt{Args} \\ &\quad \to \mathtt{Mem} \to (\mathtt{Mem} \times \mathtt{Rets})_\perp \\ \mathtt{Args}, \mathtt{Rets} &: \boldsymbol{Type} \\ \mathtt{Args} &\triangleq [\mathtt{Bits}] \\ \mathtt{Rets} &\triangleq [\mathtt{Bits}]\end{aligned}$$

L 関数 $f \in \mathtt{Lfunc}$ の意味 $\mathcal{F}[\![f]\!]$ は環境、実引数、メモリを受けとり、修正されたメモリと多値関数としての返却値の組を返す関数である．L 関数内の L 連想リストでローカルに宣言されているフレーム変数とレジスタは関数の実行中のみ存在する．フレーム変数については Interface 式によるフレームポインタのメンテナンスにより、これが実現される．レジスタについては以下の Unspecified 関数により、Prologue 式の w_r バイトだけレジスタメモリに新たなエリアを確保する．

$$\begin{aligned}&\mathtt{newregframe} : \mathtt{RMem} \to \boldsymbol{N} \to \boldsymbol{N} \\ &\mathtt{newregframe} \triangleq \text{Unspecified}\end{aligned}$$

ここに newregframe はレジスタメモリ m:RMem とバイト数 n を取り、集合 $\mathtt{RMemAddr} - \mathtt{dom}\, r$ 中の連続した n バイトをある方法で探し、そのエリアのアドレスを返す．すなわち newregframe は以下の条件を満たさなければならない．

$$\begin{aligned}&\forall x\ 0 \le x < n \to \\ &\quad \mathtt{newregframe}\, m\ n\ + x \in \mathtt{RMemAddr} - \mathtt{dom}\, r\end{aligned}$$

以下は $\mathcal{F}$ の定義である．f_newenv により関数内の環境を作り、Prologue 式を $\mathcal{E}$ で評価してフレームポインタを設定し、f_args により実引数を仮引数に代入し、f_exec により本体を実行し、f_rets により Epilogue 式にある返却値の L 式のならびを評価して多値を得、Epilogue 式を $\mathcal{E}$ で評価してフレームポインタを戻す．最後に f_newmem により、ローカルな変更 (フレームとレジスタメモリ) 以外の変更を反映させたメモリを作り、多値の返却値と組にして返す．

$$\begin{aligned}&\mathcal{F}[\![(\mathtt{FUNCTION}\ name\ alist\ seq@(pro\ \cdots\ epi))]\!] \\ &\quad \rho\,[a_1, \ldots, a_n]\,\sigma \triangleq (\sigma_6, [b_1, \ldots, b_m]) \\ &\quad \textbf{where} \\ &\quad (\mathtt{PROLOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n) \triangleq pro \\ &\quad (\mathtt{EPILOGUE}\ (w_f\ w_r)\ y_1\ \cdots\ y_m) \triangleq epi \\ &\quad \rho' \triangleq \mathtt{f\_newenv}\,\rho\,\sigma.\mathtt{rm}\ alist\ seq\ w_r \\ &\quad (\sigma_1, v_1) \triangleq \mathcal{E}[\![pro]\!]\,\rho\,\sigma \\ &\quad \sigma_2 \triangleq \mathtt{f\_args} \\ &\qquad\qquad [\![(x_1\ \cdots\ x_n)]\!]\,\rho'\,\sigma_1\,[a_1, \ldots, a_n] \\ &\quad \sigma_3 \triangleq \mathtt{f\_exec}\ seq\ \rho'\ (\sigma_2 := \{.\mathtt{pc} = 1\}) \\ &\quad (\sigma_4, [b_1, \ldots, b_m]) \\ &\qquad \triangleq \mathtt{f\_rets}[\![(y_1\ \cdots\ y_m)]\!]\,\rho'\,\sigma_3 \\ &\quad (\sigma_5, v_2) \triangleq \mathcal{E}[\![epi]\!]\,\rho\,\sigma_4 \\ &\quad \sigma_6 \triangleq \mathtt{f\_newmem}\,\sigma_5\,\sigma\end{aligned}$$

この定義中 **where** 以下の f_ で始まる関数は全てローカルであり、他からは参照されないが、組版の都合上それらを以下に分けて定義する．

- f_newenv は呼び出し側からの環境 ρ を一部修正し、L 関数でローカルな L 環境を作る．

$$\begin{aligned}&\mathtt{f\_newenv}: \mathtt{Env} \to \mathtt{RMem} \to \mathtt{Lalist} \to \mathtt{Lseq} \\ &\qquad\qquad \to \boldsymbol{N} \to \mathtt{Env} \\ &\mathtt{f\_newenv}\,\rho\ m\ a\ s\ w_r \\ &\quad \triangleq \rho := \{.\mathtt{reg} = r, .\mathtt{fra} = f, .\mathtt{lab} = l\} \\ &\qquad \textbf{where} \\ &\qquad\quad r : \mathtt{RegEnv} \\ &\qquad\quad r\,s \triangleq \mathtt{newregframe}\,m\,w_r \\ &\qquad\qquad\qquad + (\mathcal{A}[\![a]\!]\,\mathtt{REG}).\mathtt{offset}\ s \\ &\qquad\quad f : \mathtt{FrameEnv} \\ &\qquad\quad f \triangleq \rho.\mathtt{fra} := (\mathcal{A}[\![a]\!]\,\mathtt{FRAME}).\mathtt{offset} \\ &\qquad\quad l : \mathtt{LabelEnv} \\ &\qquad\quad l \triangleq \mathtt{mklabelenv}\ seq\end{aligned}$$

- f_args は引数の結合を行なう．

$$\begin{aligned}&\mathtt{f\_args}: \mathtt{InterfaceVar\ List} \to \mathtt{Env} \to \\ &\qquad\qquad \mathtt{Mem} \to \mathtt{Args} \to \mathtt{Mem}_\perp \\ &\mathtt{f\_args}[\![(x_1\ \cdots\ x_n)]\!]\,\rho\,\sigma_0\,[a_1, \ldots, a_n] \\ &\quad \triangleq \sigma_n \\ &\qquad \textbf{where} \\ &\qquad\quad t_i \triangleq x_i.\mathtt{type} \\ &\qquad\quad a_i' \triangleq \mathtt{mkconst}[\![t_i]\!]\,a_i \\ &\qquad\quad (\sigma_i, j_i) \triangleq \mathcal{E}[\![(\mathtt{SET}\ t_i\ x_i\ a_i')]\!]\,\rho\,\sigma_{i-1}\end{aligned}$$

- f_exec は Epilogue 式の直前まで実行．

$\mathtt{f\_exec}\colon \mathtt{Lseq} \to \mathtt{Env} \to \mathtt{Mem} \to \mathtt{Mem}_\perp$

$\mathtt{f\_exec}[\![s]\!]\,\rho\,\sigma$

$\triangleq$ **if** $s\,!\,\sigma.\mathtt{pc} \in \mathtt{EpilogueExpt}$ **then** σ

else $\mathtt{f\_exec}[\![s]\!]\,\rho\,(\mathcal{S}[\![s]\!]\,\rho\,\sigma\,!\,0)$

- `f_rets` は Epilogue 式の多値を評価.

$\mathtt{f\_rets}\colon \mathtt{TypedExp\ List} \to \mathtt{Env} \to$

$\mathtt{Mem} \to (\mathtt{Mem} \times \mathtt{Rets})_\perp$

$\mathtt{f\_rets}[\![(y_1\ \cdots\ y_m)]\!]\,\rho\,\sigma_0$

$\triangleq$ $(\sigma_m, [b_1, \ldots, b_m])$

where

$(\sigma_i, b_i) \triangleq \mathcal{E}[\![y_i]\!]\,\rho\,\sigma_{i-1}$

- `f_newmem` は関数呼び出し後の L メモリを、ローカルに変更されたデータメモリとレジスタメモリのドメインを元の σ のものに制限することにより作る.

$\mathtt{f\_newmem}\colon \mathtt{Mem} \to \mathtt{Mem} \to \mathtt{Mem}$

$\mathtt{f\_newmem}\,\sigma'\,\sigma$

$\triangleq$ $\sigma' := \{.\mathtt{rm} = \sigma'.\mathtt{rm}|_{\mathrm{dom}\,\sigma.\mathtt{rm}},$

$.\mathtt{dm} = \sigma'.\mathtt{dm}|_{\mathrm{dom}\,\sigma.\mathtt{dm}}\}$